Hochschule
Augsburg University of
Applied Sciences

Bachelorarbeit

Studienrichtung Technische Informatik

David Stecher
geb. Lucinkiewicz

Ein Audioplayer unter FreeRTOS in einem Low-Cost-FPGA

Verfasser der Bachelorarbeit:

David Stecher
Bahnhofstrasse 1
86653 Monheim
Telefon: 90901 / 907616
E-Mail: davidlucinkiewicz@web.de
Mat.-Nr. 1921169

Hochschule Augsburg
An der Fachhochschule 1
86161 Augsburg

Tel: +49 821 5586-0
Fax: +49 821 5586-3222
E-mail: info@hs.augsburg.de

http://www.hs-augsburg.de

Erstprüfer: Prof. Dr. Kiefer
Zweitprüfer: Prof. Dr. Högl

Beginn: 11.02.2010
Abgabe: 10.05.2010

Diese Bachelorarbeit wird unter der folgenden Creative-Commons-Lizens veröffentlicht:

http://creativecommons.org/licenses/by-nc-sa/3.0/

Inhaltsverzeichnis

1 Einführung....8
1.1 Motivation....8
1.2 Ziel dieser Arbeit....9
1.3 Aufbau der Arbeit....9
2. Entwicklungsumgebung....10
2.1 Das Spartan-3a Starter Kit....10
2.2 Der MicroBlaze Soft-Core-Prozessor....12
2.2.1 Bussystem des MicroBlaze-Prozessors....12
2.2.2 Der Aufbau des Prozessors....13
2.2.3 Interrupt-Fähigkeiten des Prozessors....13
2.2 Die EDK Werkzeugkette von Xilinx....16
3. Das Echtzeitbetriebssystem „FreeRTOS“ 18
3.1 Der Kernel....19
3.1.1 Überblick....19
3.1.2 Scheduling 19
3.1.3 Der RTOS-Tick 20
3.2 Beschreibung der FreeRTOS-Tasks und der Co-Routinen....21
3.2.1 FreeRTOS-Task....21
3.2.2 FreeRTOS-Co-Routinen....22
3.3 Synchronisation....23
3.3.1 Mutex....23
3.3.2 Semaphore....24
3.3.2.1 Binary Semaphores....24
3.3.2.2 Counting Semaphores....24
3.4 Queues....24
3.5 Die FreeRTOS-Speicherverwaltung....25
3.6 Die FreeRTOS API....26
3.6 Ein Minimales FreeRTOS-Beispielprogramm....27
3.6 Die Dateistruktur des FreeRTOS....29
3.7 Die FreeRTOS-Demoanwendung im Detail....30
3.8 FreeRTOS für ein Zielsystem konfigurieren....30
3.9 Die Konfiguration über FreeRTOSconfig.h....31
4. Entwickelte Hardware-Module....32
4.1 PCM - Master – 8....33
4.1.1 Überblick....33
4.1.2 Hardware....33
4.1.3 Interruptfunktionalität integrieren....37
4.1.4 Software-Treiber....37
4.2 „Easy and Fast“-SPI-Core....40
4.2.1 Überblick....40
4.2.2 Hardware....41
4.2.3 Simulation....44
4.2.4 Software-Treiber....45
4.3 Der „ENROT“ Dreh- / Druckknopf 46
4.3.1 Überblick....46
4.3.2 Hardware des Dreh- / Druckknopf....46

4.3.3 Simulation des „ENROT“-Druckknopfes....50
4.3.4 Simulation des „ENROT“-Drehknopfes....51
4.3.5 Software-Treiber für „ENROT“....52
4.3 LC-Display....53
4.3.1 Überblick....53
4.3.2 Hardware....54
4.3.3 Software-Treiber für das LC - Display....55
5. Der „WATOS“-Player....57
5.1 Überblick....57
5.2 Hauptprogramm....58
5.2.1 Menu-Task....59
5.2.2 Sound-Task....59
5.2.3 Error-Task....60
5.3 Verzeichnisstruktur, benötigte Dateien....61
5.4 „FAT16“- Implementierung....61
5.4.1 Überblick....61
5.4.2 „FAT16“-Aufbau:....62
5.4.3 Funktionsweise der High- und Low-Level Funktionen:....64
5.4.3.1 Die High-Level-Implementierung....64
5.4.3.2 Die Low-Level-Implementierung....67
5.5 High-und Low-Level API....68
5.6 Software Low-Level-SPI-Lösung (alternativ)....69
6. „WATOS“-Player ohne PC betreiben....69
7. Praktikumsaufgaben....70
7.1 Mögliche Aufgabenstellungen....70
8. Zusammenfassung / Fazit....73
9. Literaturverzeichnis....74
10. Anhang....76
10.1 Anleitungen und Beschreibungen....76
10.1.1 SVN-Repository....76
10.1.2 Einbinden der notwendigen IPs für FreeRTOS....77
10.1.3 Erzeugung des FreeRTOS-Tick: port.c / portasm.s....79
10.1.4 FreeRTOS für EDK-10 anpassen....80
10.1.5 Erstellen eines autonomen Systems....82
10.1.6 Timer- und Interrupt-Funktionen....84
10.1.7 SPI-Protokoll unter der Lupe....85
10.1.8 EDK-Konfigurationsdaten....87
10.1.9 Das Auslesen der FAT....89
10.2 Ressourcenverteilung....90
11. Daten-CD....93
11.1 Inhalt....93
11.2 Verzeichnisstruktur....93
12. Erklärung....93

Abbildungsverzeichnis

Abbildung 1: Spartan-3a Starter Kit....7
Abbildung 2: CoreConnect Bus....9
Abbildung 3: Interrupt-Controller Schema....11
Abbildung 4: XPS Verzeichnis-Struktur....15
Abbildung 5: Multitasking zwischen 3 Tasks [The FreeRTOS Project]....17
Abbildung 6: Scheduling zwischen 3 Tasks [The FreeRTOS Project]....17
Abbildung 7: Der RTOS-Tick [The FreeRTOS Project]....18
Abbildung 8: Mögliche Zustände eines Tasks [The FreeRTOS Project]....19
Abbildung 9: Zustände einer Co-Routine [The FreeRTOS Project]....20
Abbildung 10: Queue mit 2 Elementen....21
Abbildung 11: Interne Komponenten SoC....28
Abbildung 12: Aufbau Hardware Soundcore [SoC Ausarbeitung Florian Richter]....30
Abbildung 13: Soundcore Zustandsmaschine....32
Abbildung 14: SD-Karten Lesegerät....36
Abbildung 15: SPI-Modul Zustandsmaschine....37
Abbildung 16: Zwei-Wege Handshake Hardwaremodul....39
Abbildung 17: Simulationsergebnis SPI-Modul....39
Abbildung 18: Interner Aufbau des Drehknopfes [XILINX]....44
Abbildung 19: Der Drehknopf prellt [XILINX]....44
Abbildung 20: Zustandsmaschine für den Drehknopf....45
Abbildung 21: Simulationsergebnis des „ENROT“-Druckknopfes....48
Abbildung 22: Simulationsergebnis des „ENROT“-Drehknopfes....49
Abbildung 23: Externe Komponenten des „WATOS“-Players....54
Abbildung 24: Die Programmstruktur des „WATOS“-Players....55
Abbildung 25: „FAT16“-Aufbau....59
Abbildung 26: „FAT16“-Clusterzuordnung [22]....60
Abbildung 27: Auswahlmenü der IP-Pakte....71
Abbildung 28: Festlegen der Prioritäten....72
Abbildung 29: Adressen generieren....72
Abbildung 30: Programm Flash Memory Menü....77
Abbildung 31: Aufbau zur Timing-Analyse....79
Abbildung 32: Initialisierung der SD-Karte mit dem Befehl CMD0....80
Abbildung 33: SD-Karte antwortet mit 0x01....80
Abbildung 34: Befehl CMD1 wird gesendet....80
Abbildung 35: SD-Karte antwortet mit 0x00....81
Abbildung 36: Das Auslesen der FAT im Detail....85
Abbildung 37: MicroBlaze-IP....86
Abbildung 38: LCD-IP....87
Abbildung 39: Soundcore-IP....87
Abbildung 40: Rotary-Button-IP....87
Abbildung 41: „Easy-and-Fast“-SPI-IP....88

Tabellenverzeichnis

Tabelle 1: Programmierbare Bausteine des Spartan-3a Starter Kit und deren Konfiguration.....11
Tabelle 2: XPS-Verzeichnisaufbau.....18
Tabelle 3: FreeRTOS Schnittstellen API.....27
Tabelle 4: Einstellungen in FreeRTOSConfig.h.....31
Tabelle 5: Alle Register des Soundcores.....37
Tabelle 6: Status- u. Steuerregister im Detail.....37
Tabelle 7: Registrieren der externen Ports für das Soundmodul.....39
Tabelle 8: Register des SPI-Moduls.....43
Tabelle 9: Statusregister „ENROT“ 50
Tabelle 10: Beschaltung des LC-Displays.....54
Tabelle 11: VHDL Code Display Hardware-Modul.....54
Tabelle 12: Steuerbefehle für das LC-Display.....56
Tabelle 13: Registrieren der externen Ports für das LCD-Modul.....56
Tabelle 14: Verzeichnisstruktur „WATOS“-Player.....61
Tabelle 15: High- und Low-Level-API des Dateisystems.....68
Tabelle 16: SVN-Repository-Aufbau.....76
Tabelle 17: Ressourcenverbrauch MicroBlaze.....90
Tabelle 18: Ressourcenverbrauch Gesamtsystem.....90
Tabelle 19: Maximale Timings.....90
Tabelle 20: Inhalt der Daten CD.....93

Listings

Listing 1: Einfacher Interrupt.....14
Listing 2: Interrupt durch den XPS Interrupt-Controller.....15
Listing 3: FreeRTOS minimales Beispielprogramm.....29
Listing 4: Verzeichnisstruktur FreeRTOS.....29
Listing 5: Implementierung Core-Modul.....35
Listing 6: Erweiterung der MHS-Datei des Soundcores.....37
Listing 7: Ausschnitt aus der Testbench des SPI-Cores.....44
Listing 8: Registrieren der externen Ports für das SPI-Modul.....46
Listing 9: Ausschnitt des VHDL Codes zur Entprellung des Druckknopfes.....49
Listing 10: Ausschnitt aus der Testbench des „ENROT“-Druckknopfes.....50
Listing 11: Ausschnitt aus der Testbench des „ENROT“-Drehknopfes.....51
Listing 12: Registrieren der externen Ports für den Druck- / Drehknopf.....53
Listing 13: Steuerregister des LC-Displays.....55
Listing 14: Struktur, die der Sound-Task benötigt.....60
Listing 15: Watchdog-Implementierung.....60
Listing 16: Software-SPI Lösung.....69
Listing 17: Struktur des Queue-Eintrags.....71
Listing 18: Grundsätzlicher Aufbau des Display-Task.....72
Listing 19: Einsprung für die Interrupt-Routine.....79
Listing 20: Standard Interrupt-Routine ruft den Handler des Interrupt-Controllers auf.....79
Listing 21: Registrieren des Timers am Interrupt-Handler.....79
Listing 22: Interrupt Service Routine des Timers.....80

Listing 23: Neue Variante der prvSetupTimerInterrupt() 81
Listing 24: Neue Implementierung der Funktion vTaskISRHandler().....82
Listing 25: Compiler-Fehler im EDK 10.....82
Listing 26: Auschnitt der system.mhs.....87
Listing 27: Ausschnitt der system.ucf.....88

1 Einführung

1.1 Motivation

Mikrocontroller sind in der heutigen Zeit in fast jedem Gerät als so genannte "eingebettete Systeme" vorhanden. Sie haben die Aufgabe, das Gerät zu steuern und eine Interaktion mit dem Benutzer zu schaffen. Aktuelle Controller sind leicht zu programmieren und günstig in der Anschaffung.

Wenn es darum geht, für ein spezielles, komplexes Problem eine schnelle Lösung zu finden, wird dies meist mit einem FPGA realisiert, da dieser durch hohe Taktzeiten und Parallelverarbeitung viel schneller als ein Mikrocontroller sein kann. Außerdem lässt sich dieser auf das Problem hin optimieren, was mit einem Mikrocontroller nicht möglich ist, da sich dessen Hardware nicht mehr ändern lässt.

Durch diese Methoden ist es möglich, für viele Anwendungen entsprechende Lösungen zu entwickeln. Kombiniert man einen Mikroprozessor mit variabel gestaltbaren Peripherieeinheiten so erhält man ein „System-on-a-Chip“ (SoC). Dies stellt eine gute Kombination der vorher besprochenen Methoden dar.

Die dafür entwickelten Mikroprozessoren, wie der MicroBlaze von Xilinx, sind in IP (Intellectual-Property) Paketen verfügbar. Das bedeutet, dass das Hardwaremodul entweder als Quellcode in z.B. VHDL, oder aber auch als Netzliste verfügbar ist. Ein so erstelltes Hardwaremodul bezeichnet man auch als „Soft-Core“. Der FPGA kann dann die vorgegebene Schaltung annehmen und z.B. als Prozessor arbeiten.

Durch die Integration des Prozessors und weiterer Komponenten, wie z.B. Bussystem, Block-RAM (BRAM), Speziallogik etc. wird daraus ein SoC. Ein großer Vorteil dieser Technik ist, dass man den Prozessor und die Peripherie problemabhängig konfigurieren kann. So lässt sich der MicroBlaze Prozessor z.B. um eine Gleitkommaeinheit erweitern, wenn diese für einen besseren Ablauf benötigt wird. Dies bedeutet wiederum, dass man nicht benötigte Eigenschaften entfernen kann. Daraus resultiert ein geringer Platzverbrauch bei einer hohen Rechenleistung. So ist es möglich, eigene Hardwaremodule zu entwerfen und in das System einzubinden. Nicht zuletzt ist auch die Wiederverwendbarkeit des Hardwarecodes zu nennen, der auch für eine neue FPGA-Generation problemlos kompiliert und angewendet werden kann. Dies erspart hohe Kosten für neue Mikrocontroller und deren Softwareentwicklung.

Das konfigurierte SoC kann nun klassisch problemorientiert programmiert werden. Dies funktioniert für kleinere Projekte recht zuverlässig und ist hier noch überschaubar. Für größere, komplexere und zeitkritische Projekte bietet sich dagegen die Verwendung eines sogenannten „Echtzeitbetriebssystems“ an. In der hier vorliegenden Arbeit wird das Echtzeitbetriebssystem FreeRTOS verwendet, da es klein, kompakt, gut dokumentiert und außerdem quelloffen ist. Durch die Verwendung von FreeRTOS können Aufgaben an das Betriebssystem abgegeben werden, was den Programmierumfang verkürzt. Ein denkbarer Anwendungsfall wäre das Auslösen des Airbags in einem Auto. Sobald ein Sensor den Aufprall erkannt hat, gibt FreeRTOS durch das SoC in einer vorhersagbaren Zeit dem Airbag die Anweisung zum Auslösen.

Durch die immer häufigere Verschmelzung alltäglicher Gebrauchsgenstände mit einem SoC ist es unbedingt nötig sich mit dieser Technik auseinander zu setzen. Die vorliegende Bachelorarbeit und die vorhandenen Quellcodes können dafür einen Einstieg bieten. Mit ihrer Hilfe lässt sich eine Lernumgebung im Bereich Software, Hardware und Betriebssysteme realisieren.

1.2 Ziel dieser Arbeit

Ziel der Arbeit ist es das quelloffene Echtzeit-Betriebssystem FreeRTOS auf dem Xilinx Spartan-3a Starter Kit unter einem Microblaze-System zu installieren. Außerdem soll ein Demonstrator entwickelt werden.

Der Soundcore „PCM-Master-8“ wird als Hardwaremodul implementiert. Dieser wurde an der Hochschule Augsburg von Florian Richter und dem Autor dieser Arbeit im Rahmen des Wahlpflichtfaches "Systems-on-a-Chip" entwickelt. Für diesen Soundcore sollen FreeRTOS-kompatible Treiber entstehen. Der Soundcore muss dazu so angepasst werden, dass er Interrupts ausgeben kann.

Als Demonstration soll das Board als Audio-Player funktionieren. Über eine SD-Karte werden Musiktitel auf dem Display aufgelistet, diese können dann mit dem Drehknopf angewählt und somit abgespielt werden. Die SD-Karte wird vom SoC mit dem Serial Peripheral Interface (SPI) angesprochen, das dafür nötige Hardwaremodul wird selbst entworfen.

Für die Realisierung des Demonstrators werden außerdem Treiber für Display, Drehbutton und SPI-Modul entwickelt. Das System wird mit einem Bootloader versehen, daher wird es autonom lauffähig sein.

Des Weiteren soll mit diesem Projekt eine Lernplattform für andere Studenten im Bereich SoC und Betriebssysteme entstehen.

1.3 Aufbau der Arbeit

Das zweite Kapitel stellt die Werkzeuge vor, mit denen die Software und Hardware erstellt wurde. Dabei werden die grundlegenden Funktionen besprochen.

Das dritte Kapitel befasst sich mit FreeRTOS, dabei werden ein einfaches Grundprogramm, die Demoanwendung, die Konfigurationsmöglichkeiten, Schnittstellen, API Funktionen und grundlegende Inhalte aufgezeigt.

Im vierten Kapitel werden alle vom Autor dieser Arbeit erzeugten Hardware-Module, sowie Software-Treiber und Simulationsberichte erklärt.

Das fünfte Kapitel befasst sich mit dem Demoprogramm "WATOS-Player". Dabei werden die Vorzüge von FreeRTOS aufgezeigt.

Das sechste Kapitel zeigt, wie man ein fertiges Projekt autonom lauffähig auf das Spartan 3a überträgt. Die dabei notwendigen Schritte werden im Anhang genau besprochen. Damit ist es möglich, das Board ohne vorherige Programmierung zu betreiben.

Im siebten Kapitel werden mögliche Praktikumsaufgaben gestellt. Die Aufgaben sind so formuliert, dass ein Bearbeiter möglichst viel im Bereich Software, Hardware (VHDL) und Treiberprogrammierung lernt, um einen größtmöglichen Einblick in die Funktion eines eingebetteten Systems zu bieten.
Schließlich findet sich im achten Kapitel die Zusammenfassung.

2. Entwicklungsumgebung

2.1 Das Spartan-3a Starter Kit

Abbildung 1 zeigt das Spartan-3a Starter Kit. Das Board bietet eine Vielzahl an Schnittstellen zur Außenwelt. Die für die Arbeit wichtigen Anschlüsse sind zuerst der USB-Anschluss zur Programmierung des FPGA oder der Flash-Speicherbausteine, für Debug-Zwecke wird eine der seriellen Schnittstellen verwendet. Das SD-Karten-Lesegerät wird an einem Erweiterungsstecker angeschlossen. Alle anderen Schnittstellen werden nicht benutzt.

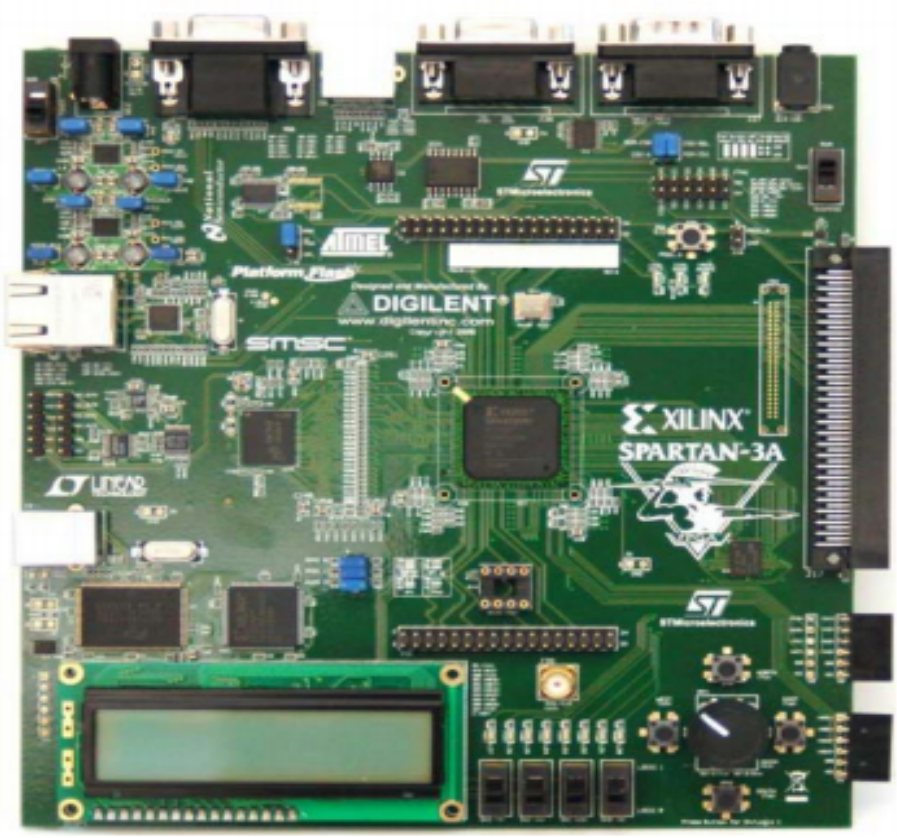


Abbildung 1: Spartan-3a Starter Kit

Das Spartan-3a Board im Überblick [2] :

Kernkomponente des Starter Kit ist der Xilinx Spartan-3A (XC3S700A-FG484) FPGA in der Mitte des Boards. Die Ressourcen-Ausnutzung dieses Bausteines ist im Anhang beschrieben. Ein weiterer Xilinx-Baustein befindet sich oben links unter dem VGA-Anschluss. Es handelt sich hierbei um den Platform-Flash-Baustein (XCF04S-VOG20C). Dieser wird verwendet, um die Hardwareinformationen des FPGA zu speichern. Das heißt, der FPGA kann durch diesen Chip konfiguriert werden. Das Kapitel 6 "WATOS-Player ohne PC betreiben" befasst sich detailliert mit diesem Thema. Durch den Konfigurationsmodus wird eingestellt, wie der FPGA beschrieben werden soll. Geschieht dies z.B. durch den PC mit der USB-Schnittstelle, so lautet der Modus JTAG.

Den gewünschten Modus stellt man durch die Jumper M1 - M3 ein, wobei eine Null für „Jumper gesteckt“ steht. Eine Übersicht über alle programmierbaren Bausteine und ggf. deren Konfigurationsmöglichkeiten gibt die Tabelle 1.

Bauteil:	Beschreibung:	Konfigurations-modus:	Jumperstellung M2:M1:M0
4 Mbit Platform Flash PROM (500 KByte)	Platform Flash (XCF04SVOG20C) speichert den Bitstream.	Master Serial	0:0:0
DDR2 SDRAM (65 MByte)	Hauptspeicher Anwenderprogramm	-	-
32 Mbit parallel Flash (4 MByte)	Hier findet das Anwenderprogramm platz. Es wird nach dem Einschalten des FPGA Boards vom Bootloader in den DDR2 RAM kopiert.	Master BPI Up •	0:1:0
2 x 16 Mbit SPI Flash Device (2 MByte)	Hier könnte man das Anwenderprogramm auch abspeichern. Wegen der leicht zu ansprechenden Schnittstelle (SPI) wird es auch zur Speicherung von Programmdaten benutzt.	Master SPI	0:0:1
XC3S700A-FG484 Spartan 3a	Kernkomponente, beinhaltet Hardwaresystem	JTAG	1:0:1

Tabelle 1: Programmierbare Bausteine des Spartan-3a Starter Kit und deren Konfiguration

Weiterhin verwendet werden das LC-Display, die Druckknöpfe und der Drehknopf. Natürlich bietet das Board weitaus mehr Schnittstellen bzw. Komponenten, die aber in dieser Arbeit nicht verwendet werden. Genauere Informationen kann man im Datenblatt des Starter-Kit unter [2] beziehen.

2.2 Der MicroBlaze Soft-Core-Prozessor

2.2.1 Bussystem des MicroBlaze-Prozessors

Das mit dem MicroBlaze ausgestattete SoC verwendet als Bussystem CoreConnect von IBM [9]. CoreConnect ist ein Mikroprozessorbus, der speziell für SoC Systeme entwickelt wurde. Abbildung 2 zeigt den Aufbau eines solchen Busses.

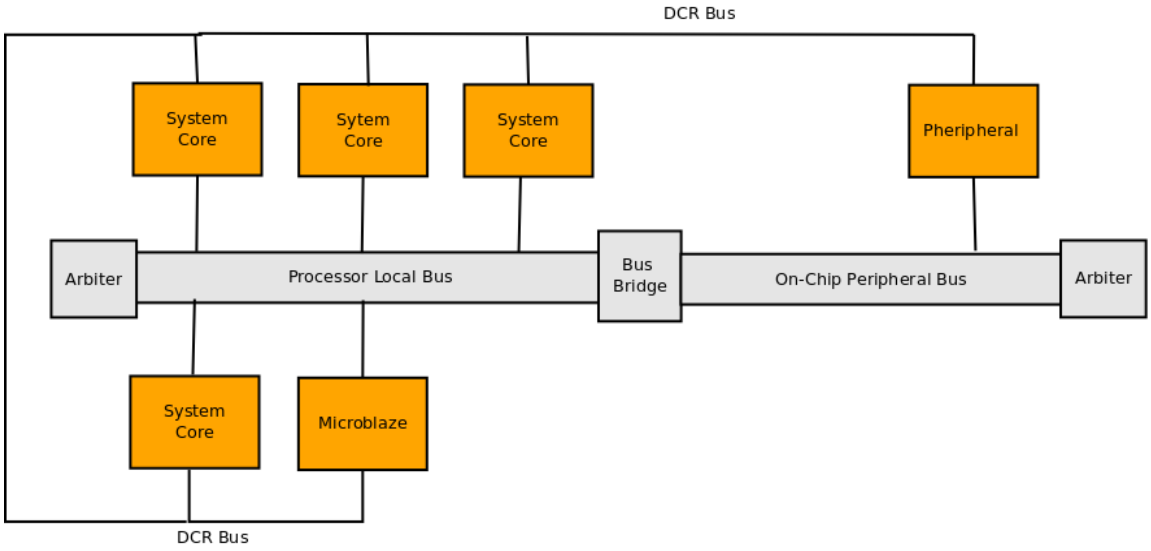


Abbildung 2: CoreConnect Bus

Wie aus Abbildung 2 ersichtlich, setzt sich CoreConnect aus drei verschieden Bussen zusammen. Das sind der PLB ("Processor Local Bus"), der OPB ("On-Chip Peripheral Bus") und der DCR ("Device Control Register Bus").

PLB ("Processor Local Bus")

Als Standardbus im EDK 10 dient der PLB [14]. Er bietet eine Unterstützung für mehrere Master, die Anfragen an den Bus stellen dürfen. Ein flexibler Bus-Arbiter kümmert sich um die korrekte Abarbeitung der Anfragen. Der unter 4.1 besprochene Soundcore PCM-Master-8 arbeitet neben dem MicroBlaze-Prozessor ebenfalls als Master. Es sind variable Busbreiten von 32-Bit bis 128-Bit realisierbar. Für Schreib- und Lesezugriffe sind 2 separate Busse implementiert, daraus resultiert eine höherer Datendurchsatz. Das Bus-Design ist komplett synchron entwickelt worden.

Der PLB Bus zeichnet sich durch eine hohe Leistung aus, die durch Burst-Transfers, geteilte Transaktionen ("split transactions"), Pipelining, sowie mögliche überlappende Arbitrierung (Busvergabe schon während einer noch laufenden Transaktion) erreicht wird.

OPB ("On Chip Peripheral Bus")

Der OPB Bus bietet, wie sein schnelleres Pendant, die Unterstützung mehrerer Master, separate Busse zum Lesen und Schreiben, einen synchroner Entwurf und die Möglichkeit von Burst-Transfers. Der OPB bietet einen Adress- und Datenbus von maximal 32 Bit. Er ist von Design her weniger flächenaufwendig als der PLB, aber auch weniger performant. Es bietet sich nicht an, den

Prozessor direkt über OPB zu betreiben, da dieser Bus zu langsam ist. Der Bus ist für die Kommunikation mit langsamerer Hardware konzipiert. Damit der Prozessor die Peripherie des OPB ansprechen kann, gibt es eine „PLB to OPB bridge unit“. Ein Master am OPB kann durch eine „OPB to PLB bridge unit“ mit dem PLB-Bus kommunizieren (siehe Abbildung 2).

DCR ("Device Control Register Bus")

Der 32 Bit DCR-Bus ist ein sehr einfach gehaltener Bus mit minimalem Logikaufwand. Er ist für die Verwendung speziell auf Geräteregister konzipiert. Der Durchsatz ist daher unbedeutend.

Ein weiteres im SoC vorkommendes Bussystem ist der LMB („Local Memory Bus“):

Der LMB-Bus ist ein schneller lokaler Bus um die MicroBlaze Instruktions- und Daten-Ports über den „XPS_BRAM Interface Controller“ an den On-Chip BRAM anzubinden. Er ist komplett synchron entworfen. Für Instruktionen und Daten werden zwei separate Busse verwendet.

2.2.2 Der Aufbau des Prozessors

MicroBlaze ist ein in FPGAs der Firma Xilinx verwendbarer Mikrocontroller. Dieser Mikrocontroller existiert nicht als physische Hardware, sondern ist nur als Soft-Core in Form einer Netzliste verfügbar.

Durch die spezielle Optimierung auf die Besonderheiten bestimmter FPGA-Bausteine ist der Logikbedarf dieses Mikrocontrollers gering und bewegt sich je nach Ausbaustufe und Version zwischen rund 700 und über 2000 Slices (kleinste logische Funktionseinheit).

MicroBlaze ist ein 32-Bit-RISC-Mikroprozessor, der in der Architektur dem DLX-Mikroprozessor [23] ähnelt. Er besitzt eine konfigurierbare drei- bis fünfstufige Pipeline, internen Cache, verfügt über einen Interrupt, einen hardwarebasierenden Multiplizierer und (optional) eine hardwarebasierende Divisionseinheit, eine Gleitkommaeinheit, einen Barrel-Shifter und spezielle Schieberegistereinheiten.

Er besitzt außerdem mehrere unterschiedliche Busse, welche für den Anschluss von umfangreicher Peripherie und Speicher in einem FPGA vorgesehen sind. Der Prozessor ist an einem FPGA-internen PLB-Bus der CoreConnect-Spezifikation [14] angebunden. Zusammen mit einem optionalen externen Speicher und weiteren Peripherieeinheiten am PLB-, OPB- oder DCR-Bus entsteht ein System-on-a-Chip.

2.2.3 Interrupt-Fähigkeiten des Prozessors

Der MicroBlaze besitzt einen externen Interrupt-Port. Interrupts werden nur beachtet, wenn das Interrupt-Enable-Bit (IE) im Status-Register gesetzt ist. Geschieht ein Interrupt, wird der momentan in Ausführung befindliche Befehl abgearbeitet, während der in der Dekodierungsphase befindliche Befehl durch einen Sprung zum Interrupt-Vector (0x10) ersetzt wird. Die in Ausführung befindliche Interrupt-Service-Routine kann nicht mehr unterbrochen werden. Die Rücksprungadresse wird in das Register R14 gespeichert [4].

Listing 1 zeigt den Beispielcode eines implementierten Interrupt-Handlers, der genau auf einen Interrupt reagieren kann. Die mit `__attribute__((interrupt_handler))` deklarierte Funktion wird ausgeführt, sobald ein Interrupt ausgelöst wurde.

```
void my_ISR(void) __attribute__((interrupt_handler));

int main(void){
microblaze_enable_interrupts()
…}

void my_ISR(void){
microblaze_disable_interrupts()
//  clear ALL interrupt sources //  your ISR Code
microblaze_enable_interrupts()
}
```

Listing 1: Einfacher Interrupt

Erweiterung der Interrupt-Fähigkeit durch den XPS-Interrupt-Controller

Da der MicroBlaze nur einen Interrupt-Port bietet, aber mehrere Interrupts ausgewertet werden müssen, wird ein Interrupt-Controller benötigt. Dabei wird der XPS-Interrupt-Controller zwischen den max. 32 Quellen und dem Interrupt Eingang des MicroBlaze geschaltet. Abbildung 3 zeigt das Anschlussschema.

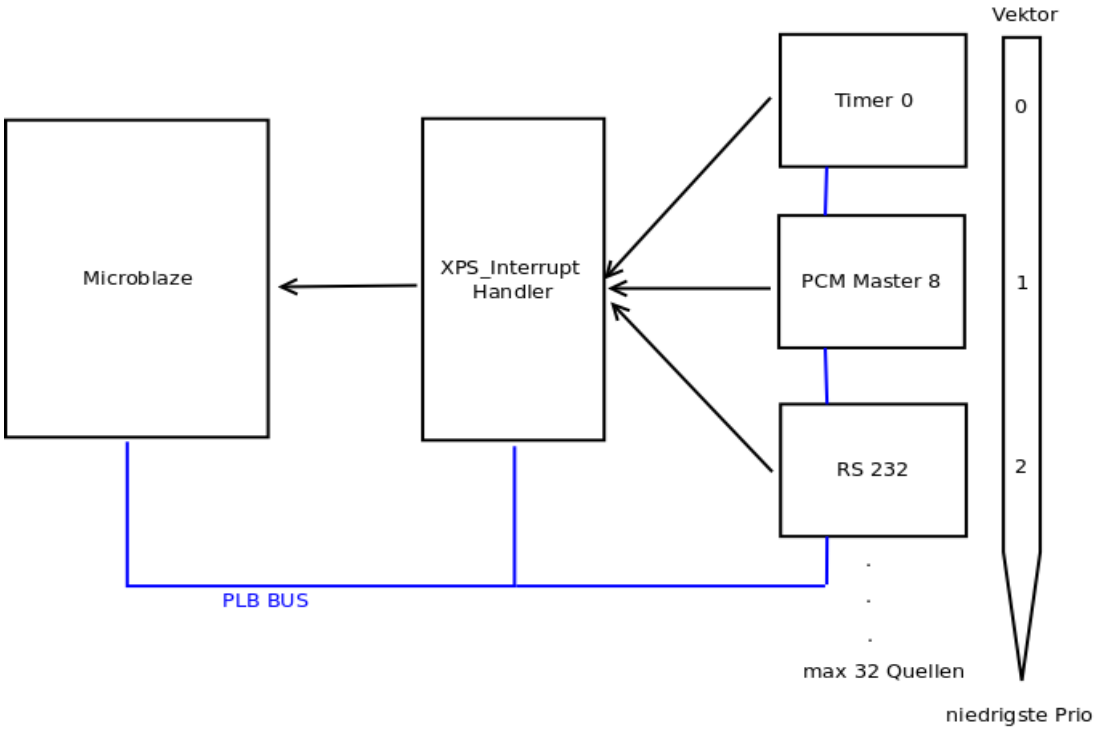


Abbildung 3: Interrupt-Controller Schema

Listing 2 zeigt den erhöhten Aufwand an Codezeilen. Jedes Gerät, das einen Interrupt ausgeben kann, muss vorher am Interrupt-Handler angemeldet werden und bekommt seine eigene Interrupt-Service-Routine.

```
int main (void)
{
// register all ISR's to INTC
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                      XPAR_XPS_INTC_0_PCM-Master-8_INTERRUPT_INTR,
                     (XInterruptHandler) pcm8_ISR,
                     (void *)XPAR_PCM8_BASEADDR);

XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                      XPAR_XPS_INTC_0_TIMER_1_INTERRUPT_INTR,
                     (XInterruptHandler) timer_ISR,
                     (void *)XPAR_XPS_TIMER_1_BASEADDR);

XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
                      XPAR_XPS_INTC_0_RS232_INTR,
                     (XInterruptHandler)rs232_ISR,
                     (void *)XPAR_RS232_BASEADDR);
// Enable INTC
microblaze_enable_interrupts();
while (1);  // wait for interrupt
return 0;
}
void timer_ISR (void)
{
microblaze_disable_interrupts();
// clear ALL interrupt sources,
// your Code
microblaze_enable_interrupts();
}
```

Listing 2: Interrupt durch den XPS Interrupt-Controller

Ablauf eines Ein Interrupts durch den Interrupt-Controller [7][8]:

1. Der Timer verursacht einen Interrupt
2. Der INTC empfängt den Interrupt und leitet ihn an den MicroBlaze weiter
3. Wenn das Interrupt Enable Bit gesetzt ist, reagiert der MicroBlaze und führt die ISR mit dem Namen `_interrupt_handler()` aus, außerdem wird das IE-Bit wieder gelöscht.
4. Es wird geprüft welche Funktion als Interrupt-Handler eingetragen ist, in unserem Fall die `XIntc_DeviceInterruptHandler()` unseres Interrupt-Controllers.
5. Diese Funktion prüft, welche Geräte einen Interrupt verursacht haben. Sie wählt das mit der höchsten Priorität aus und ruft die eingetragene Funktion `timer_ISR (void)` auf.
6. Die `timer_ISR (void)` wird nun ausgeführt.
7. Danach quittiert `XIntc_DeviceInterruptHandler()` den Interrupt und prüft, ob noch weitere Interrupts auszuführen sind. Wenn nicht, wird beendet und zur Funktion `_interrupt_handler()` zurückgekehrt.
8. Diese wird ebenfalls verlassen, das IE-Bit wird wieder gesetzt, der Kontext wird zurück gespeichert.

Die Realisierung unter FreeRTOS findet sich im Kapitel 9.1.3 .

2.2 Die EDK Werkzeugkette von Xilinx

Embedded Development Kit

Das Embedded Development Kit ist eine Sammlung von Hilfsprogrammen und Hardwaremodulen (Intellectual Propertys), die es ermöglicht, ein komplettes, eingebettetes Prozessor-System in einen Xilinx-FPGA zu implementieren. Dabei enthält sie alle benötigten Werkzeuge und vorkompilierten Komponenten für verschiedene Entwicklungsboards (Spartan 3a, 3e, Virtex5,..), um ein Embedded-System zu entwerfen. Einige dieser Komponenten sind: der MicroBlaze-Prozessor selbst, Timer-Controller, der vorher besprochene Interrupt-Controller, oder das im Kapitel 2.2.1 erklärte Bussystem. Um Xilinx-EDK auszuführen, wird Xilinx-ISE benötigt [1].

Software Development Kit

Das Platform Studio Software Development Kit (SDK) ist eine zu XPS komplementäre Entwicklungsumgebung, in der man mit C / C++ Programme entwerfen kann. SDK basiert auf dem Eclipse-Framework [1].

Xilinx Platform Studio

Ein weiteres Werkzeug ist das Xilinx Platform Studio (XPS). Es ist eine grafische

Entwicklungsumgebung, um die Hardware, sowie die Software des Embedded Systems einzurichten, bzw. zu programmieren. Dabei wird auf die GNU Compiler Collection zurückgegriffen [1]. XPS enthält Assistenten die dem Benutzer helfen ein komplettes SoC zu konfigurieren, eigene IP-Cores zu erstellen, oder Bestehende hinzuzufügen. Dank der grafischen Darstellung lassen sich komfortabel Verbindungen herstellen, oder Peripherieeigenschaften editieren.

Zuerst wird bei einem Projekt die Hardware samt Peripherie erstellt. Daraus resultiert die Datei `system.bit`. Danach folgt die Entwicklung der Software entweder mit XPS oder SDK. Diese wird kompiliert und mit den Hardwaredaten in die `download.bit` Datei gepackt. Mit dieser Datei es es jetzt möglich, den FPGA zu programmieren. Mithilfe von IMPACT kann man die Datei strukturell so ändern, dass sie auf einen Flash Speicher geschrieben werden kann. Dieser initialisiert dann beim Einschalten das FPGA.

Abbildung 3 zeigt den typischen Verzeichnisaufbau eines XPS-Projektes. Die zwei vorher beschriebenen `.bit` Dateien befinden sich in dem Verzeichnis `Implementation`, das aber erst nach erfolgreicher Synthese erstellt wird.

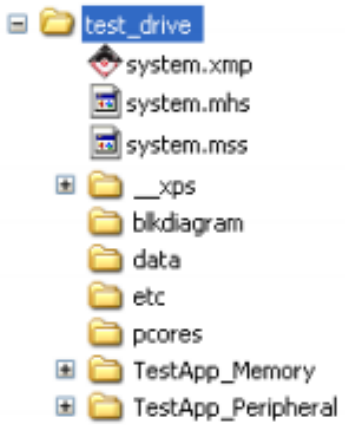


Abbildung 3: XPS-Verzeichnis-Struktur

Tabelle 2 zeigt die Inhalte der Verzeichnisse und Konfigurationsdateien.

__xps	Projektdaten
data	system.ucf: Zuordnung der Hardware Ausgänge zu den realen FPGA Pins
etc	aufgenommene Konfigurationsdaten
pcores	selbst erstellte Hardware
system.xmp	Hauptprojektdatei
system.mhs	Hardware-Spezifikation aller Systemkomponenten eins. Parameter
system.mss	Software-Spezifikation aller Systemkomponenten eins. Parameter

Tabelle 2: XPS-Verzeichnisaufbau

3. Das Echtzeitbetriebssystem „FreeRTOS“

Ein RTOS bietet Echtzeitfunktionalität, was für die unbedingte Einhaltung von Zeitbedingungen und Vorhersagbarkeit des Prozessverhaltens steht.

FreeRTOS ist ein Open-Source Echtzeit Betriebssystem (RTOS), das für verschiedene Mikrocontroller (AVR, Freescale, Hitachi H9, Microblaze, PowerPC,..) verfügbar ist und ständig erweitert wird. Entwickelt wurde es von Richard Barry für derzeit 20 Plattformen. Da der Sourcecode hauptsächlich in C geschrieben wurde, ist es höchst portabel. Es steht unter GPL Lizenz [25] und kann somit in eigenen Projekten verwendet werden.

Zu dem Projekt gehören auch die nicht quelloffenen Pendants SafeRTOS und OpenRTOS. Sie sind funktional identisch und verwenden die gleiche API, wobei das SafeRTOS eine vom TÜV Süd bis Sicherheitsstufe SIL 3 [24] zertifizierte Version des kommerziellen OpenRTOS ist.

FreeRTOS ist ein sehr minimalistisches Betriebssystem. Es bietet grundlegende Scheduler-Funktionen, Interprozesskommunikation (IPC) und Semaphore für die Synchronisation. Benötigt man einen großen Funktionsumfang, Ein- und Ausgabe, Netzwerkfunktionen, etc. so sollte man z.B. auf das RTOS RTLinux zurückgreifen. Es ist allerdings wegen des Linux Kernels um einiges größer.

FreeRTOS bietet aufgrund seiner Größe auch viele Vorteile. Es verbraucht sehr wenig Ressourcen. Ein typisches Kernel-Image hat zwischen 4Kb und 9Kb. Der Kernel ist relativ einfach aufgebaut, er besteht aus nur 3 Dateien. Eigene Weiterentwicklungen scheitern daher nicht an der unüberschaubaren Flut von Code-Dateien. Nicht zuletzt bietet FreeRTOS eine sehr umfangreiche Dokumentation, die einen Einstig erleichtert.

3.1 Der Kernel

3.1.1 Überblick

Jedes ausführbare Programm in FreeRTOS ist ein Task (oder eine Co-Routine), der unter der Kontrolle des Kernels (genauer Scheduler) steht. Wenn ein Kernel mehr als einen Task gleichzeitig ausführen kann, redet man von Multitasking. Abbildung 5 zeigt das Multitasking zwischen 3 Tasks. Alle Tasks haben in diesem Fall eine gleich große Zeitscheibe. Der FreeRTOS-Kernel kann entweder „preemtive“ oder „cooperativ“ betrieben werden. Bei der ersten Betriebsart entscheidet der Scheduler mithilfe von Timer-Interrupts, wann ein Task die Ausführung abgibt. Bei der Zweiten entscheidet der Task selbst, wann er die Rechenzeit abgibt.

Zwischen den Tasks kann man entweder mit einer Queue kommunizieren, oder mit speziellen Synchronisationsverfahren durch Semaphore oder Mutexe. Bearbeitet ein Task einen kritischen Abschnitt (Betriebsart: „preemtive“), der nicht unterbrochen werden darf, kann dieser mithilfe spezieller Kernel-Funktionen gesichert werden.

FreeRTOS bietet 3 verschiedene Modelle, um Speicher dynamisch vom Heap anzufordern. Jedes Modell befindet sich in einer eigenen Datei. Es kann nur ein Modell gleichzeitig verwendet werden.

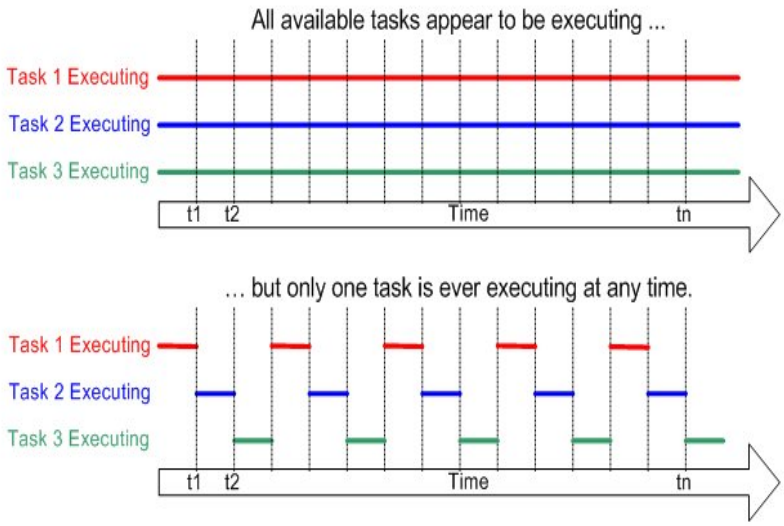


Abbildung 5: Multitasking zwischen 3 Tasks [10]

3.1.2 Scheduling

Der Scheduler ist der Teil des Kernels, der entscheidet, welcher Task momentan ausgeführt wird. Ein Task kann im Laufe seines Bestehens vom Scheduler beliebig oft gestartet und angehalten werden. Ausgeführt wird immer der Task mit der höchsten Priorität, es sei denn, er wartet auf eine belegte Ressource. Existieren zwei Tasks mit der gleichen Priorität, kommt das Round-Robin [26] Verfahren zum Einsatz. Befindet sich ein Task in einem kritischen Abschnitt, der durch den Scheduler nicht unterbrochen werden darf, kann dieser Abschnitt entweder mithilfe der Kernel-

Funktion `taskENTER_CRITICAL` geschützt werden, oder der Task suspendiert mit der Kernel-Funktion `VtaskSeuspendAll()` den Scheduler für eine gewisse Zeit. Letzteres hat den Vorteil, dass Interrupts immer noch empfangen werden können. Wenn ein Task den exklusiven Zugriff auf eine Ressource benötigt, kann er diese entweder mit einem Mutex, oder einem Semaphore schützen.

Der Task kann nicht nur durch den Scheduler angehalten werden. Er kann sich selbst schlafen (sleep) legen, um eine gewisse Zeit zu verzögern, oder er geht in den Wartezustand (block), wenn er auf eine Ressource zugreifen will, die momentan belegt ist. Ein Task, der in einem der beiden Wartezustände ist, wird nicht ausgeführt und bekommt daher keine Rechenzeit. Abbildung 6 zeigt ein solches Verhalten.

Zeitlicher Ablauf der Abbildung 6:

- Task A wird ausgeführt (1)
- Task A wird angehalten (2)
- Task 2 wird gestartet (3)
- Task 2 sperrt eine Ressource für seinen exklusiven Zugriff bei der Ausführung (4)
- Task 2 wird angehalten (5), Task 3 wird gestartet (6)
- Task 3 will auf die gleiche Ressource wie Task 2 zugreifen, da sie aber gesperrt ist beendet sich Task 3 selbständig (7)
- Task 1 wird gestartet (8)
- Task 2 (9) beendet den Zugriff auf die Ressource und gibt sie wieder frei
- Task 3 (10) kann nun auf die Ressource zugreifen.

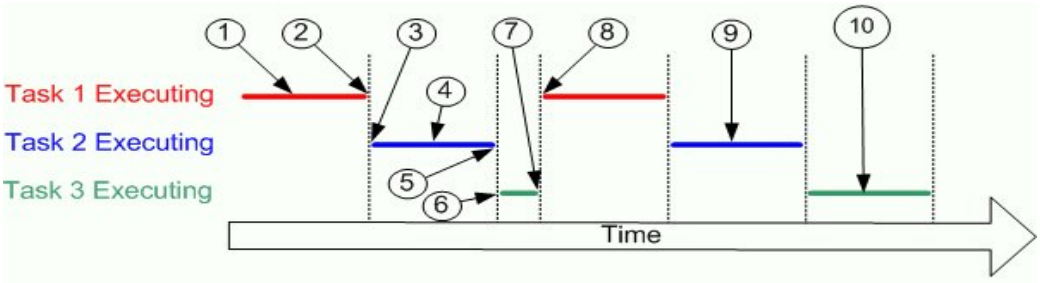


Abbildung 6: Scheduling zwischen 3 Tasks [10]

3.1.3 Der RTOS-Tick

Wenn ein Task im "sleep" Modus ist, gibt er an, wann er wieder aufwachen möchte. Wenn ein Task im "block" Modus ist, gibt er eine maximale Zeit an, die er bereit ist zu warten. In beiden Fällen wird keine CPU Zeit verbraucht.

Der FreeRTOS Kernel verwendet für das Messen der Zeit eine "tick" variable. Ein Timer Interrupt inkrementiert die "tick" Variable. Die Zeitauflösung ist also minimal ein "tick". Jedesmal, wenn der "tick" Zähler inkrementiert wird, muss der Kernel prüfen, ob die Rechenzeit neu vergeben werden

muss. Einen beispielhaften Ablauf zeigt die Abbildung 7.

Ein so ausgeführter Kontextwechsel wird "preemptive" (präemptiv) genannt, da der unterbrochene Task seine Ausführung nicht freiwillig gestoppt hat.

Durch die Makros `taskENTER_CRITICAL()` bzw. `taskEXIT_CRITICAL` lassen sich Bereiche erzeugen, die nicht von Interrupts unterbrochen werden können. Dies ist speziell bei der Programmierung von Treibern wichtig.

Zeitlicher Ablauf der Abbildung 7:

- Der Idle-Task wird ausgeführt (1)
- Der RTOS Tick inkrementiert die "tick"Variable, die `TickISR()` wird ausgeführt (3)
- Die ISR bringt `vControlTask` zur Ausführung, da dieser eine höhere Priorität als der `Idle-Task` hat. Ein Kontextwechsel wird ausgeführt.
- `VControl-Task` beginnt die Ausführung (5)

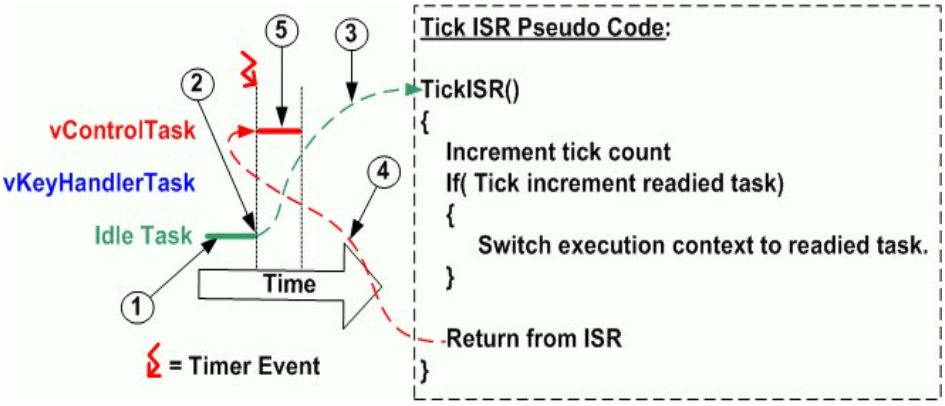


Abbildung 7: Der RTOS-Tick [10]

3.2 Beschreibung der FreeRTOS-Tasks und der Co-Routinen

3.2.1 FreeRTOS-Task

Die wohl wichtigste Eigenschaft eines Tasks ist dessen Priorität. Sie gibt Auskunft darüber, wie der Scheduler den Task bearbeitet. Jeder Task besitzt einen eigenen Stack und wird in seinem eigenen Kontext ausgeführt. Es besteht dabei keine Abhängigkeit zu anderen Tasks oder zum Scheduler. Es gibt einen Idle-Task, der immer dann ausgeführt wird, wenn kein anderer Task zur Bearbeitung da ist. Er gibt die Ressourcen der gelöschten Tasks frei. Außerdem kann er eine Callback-Funktion aufrufen: "idle-hook". In dieser Funktion könnte der Prozessor z.B. in einen Stromsparbetrieb geschalten werden. Ein Task wird mit der Funktion `xTaskCreate()` erstellt. Er kann sich in einem von vier Zuständen befinden (siehe Abbildung 8).

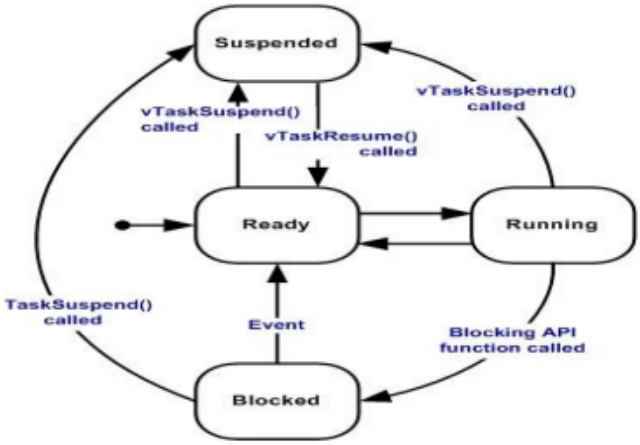


Abbildung 8: Mögliche Zustände eines Tasks [10]

Ready: Bereit zum Ausführen; wartet, bis Scheduler CPU zuteilt

Runnig: Wird momentan ausgeführt

Suspended: Mit `vTaskSuspend()` kann man einen Task explizit suspendieren, es gibt keinen max. Timeout-Wert. Task wird erst wieder mit `XTaskResume()` "ready" geschaltet.

Blocked: Wartet auf internes oder externes Ereignis, z.B. Zugriff auf die Queue oder abgelaufene Verweilzeit. Es gibt einen maximalen Timeout-Wert, nach dem der Task wieder "ready" werden kann.

3.2.2 FreeRTOS-Co-Routinen

Co-Routinen haben keinen eigenen Stack. Daher eignen sie sich für sehr kleine Systeme. Sie teilen sich den Stack mit dem Task, in dem sie aufgerufen wurden. Auch Co-Routinen haben eine bestimmte Priorität, sie gilt ausschließlich für die Routine und ist nicht auf die Tasks bezogen. Eine Co-Routine wird mit `crSTART(xHandle)` und `crEND(xHandle)` aufgerufen. Das Scheduling erfolgt durch wiederholtes Aufrufen der Funktion `vCoRoutineSchedule()`. Ein geeigneter Ort für den Aufruf dieser Funkton ist die Idle-Hook Funktion, die durch den IDLE-Task gesteuert wird. Ein Task hat daher immer eine höhere Priorität als eine Co-Routine. Abbildung 9 zeigt die Zustände einer solchen Routine.

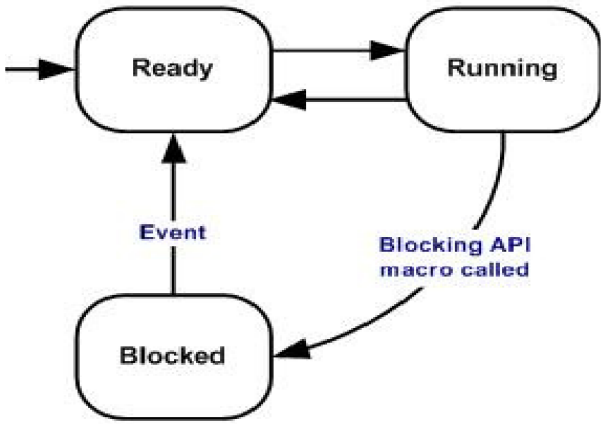


Abbildung 9: Zustände einer Co-Routine [10]

Ready: Bereit zum Ausführen; wartet, bis Scheduler CPU zuteilt

Running: Wird momentan ausgeführt

Blocked: Wartet auf internes oder externes Ereignis.

3.3 Synchronisation

3.3.1 Mutex

Mutexe (mutal exclusion – gegenseitiger Ausschluss) werden verwendet, um eine bestimmte Ressource für den exklusiven Zugriff zu schützen. Ein Mutex ist im Prinzip nichts anders als eine Queue mit einem Eintrag. Es interessiert jedoch nicht der Eintrag der Queue, sondern nur, ob er da ist oder nicht. Erstellt wird ein Mutex mit der Funktion `xSemaphoreCreateMutex()`. Will ein Task auf ein Gerät zugreifen, prüft er mit `xSemaphoreTake()` ob das Gerät geschützt ist (Queue ist leer). Ist dies der Fall, geht der Task in den „blocked“ Mode. Ist der Mutex jetzt frei geworden (Queue ist voll), kann der Task auf das Gerät zugreifen (die Nachricht entnehmen). Das Gerät ist wieder geschützt, da es über den Mutex vom Task für seinen Zugriff gesperrt wurde. Ist der Task mit seiner Abarbeitung fertig, signalisiert er dies mit `xSemaphoreGive()` (er schickt eine Nachricht in die Queue) und das Gerät ist wieder frei. *Nur der Task, der den Mutex belegt, kann ihn wieder freigeben.*

3.3.2 Semaphore

3.3.2.1 Binary Semaphores

Ein Binary Semaphore ist im Prinzip das Gleiche wie ein Mutex, mit einem Unterschied: *Jeder Task kann das Semaphore freigeben, unabhängig davon ober er es gesperrt hat oder nicht*. Binary Semaphores können also ein Ereignis signalisieren. Ähnlich einer Ampel: Ist diese grün, können alle Autos losfahren. Erstellt wird ein Binary Semaphore mit der Funktion `vSemaphoreCreateBinary()`. Genau wie bei Mutexen wird das Semaphore mit der Funktion `xSemaphoreTake()` abgefragt.

Beispiel:

Ein Interrupt signalisiert die Freigabe einer Ressource (Ampel ist grün). Der nächste Task, der aktiv wird holt sich das Semaphore (Ampel wird rot) und arbeitet an der Ressource, bis er fertig ist.

3.3.2.2 Counting Semaphores

Counting Semaphores werden entweder verwendet, um Ereignisse zu zählen, oder um mehrere Ressourcen zu managen. Die Funktion `vSemaphoreCreateCounting()` erstellt eine solches Semaphore. Abgefragt wird es ebenfalls mit der Funktion `xSemaphoreTake()`.

Im ersten Fall wird bei jedem Ereignis eine Zähl-Variable inkrementiert. Diese wird bei jedem „Entnehmen“ des Semaphores wieder dekrementiert. So kann kontrolliert werden, auf wie viele Ereignisse schon reagiert wurde. Die Variable ist in diesem Fall am Anfang Null.

Im zweitem Fall zeigt die Zähl-Variable die freien Ressourcen an. Erhält ein Task ein Semaphore, dekrementiert er den Zählerstand. Gibt er das Semaphore ab, wird der Zählerstand wieder inkrementiert. Die Variable ist also am Anfang mit der Anzahl der freien Ressourcen initialisiert. Eine gute Erklärung zum Unterschied zwischen Mutexe u. Counting Semaphores ist unter [27] zu finden.

3.4 Queues

Queues können zwischen Tasks oder Interrupt-SR und Tasks aufgebaut werden. Eine Synchronisation durch die Anwendung ist nicht notwendig. Queues benutzen einen Thread-sicheren FIFO Puffer, in den Nachrichten hineingeschickt werden können. Der Task auf der anderen Seite kann sich die Nachricht dann abholen. Geschickt wird nur eine lokale Kopie der Nachricht um zu verhindern, dass beide Tasks gleichzeitig auf die selbe Nachricht zugreifen. Abbildung 10 zeigt eine Queue, die 5 Einträge groß ist.

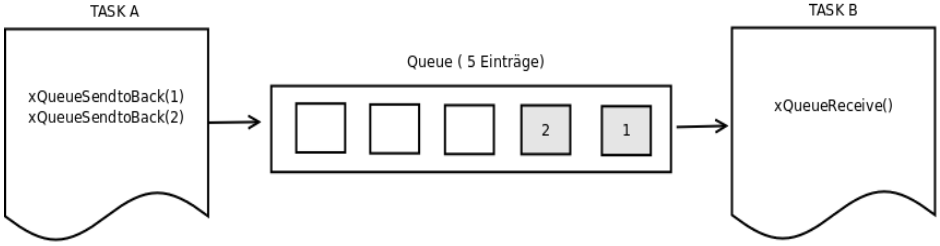


Abbildung 10: Queue mit 2 Elementen

Eine Queue kann auf folgende Weisen verwendet werden:

Der Empfangstask hat eine höhere Priorität als der Sendetask:

Das bedeutet, dass niemals mehr als eine Nachricht in der Queue bereit steht. Denn sobald eine Nachricht da ist, geht der Empfangs-Task in den aktiven Zustand, blockiert den Sende-Task und holt sich die Nachricht aus der Queue. Der Empfangstask kann eine Zeit, "block time", warten, bevor er wieder in den "blocked state" geht.

Der Sendetask hat eine höhere Priorität als der Empfangstask:

Jetzt sind die Prioritäten genau umgekehrt. Dass heißt, die Queue ist immer voll. Denn wenn der Empfangstask eine Nachricht entnommen hat, wird er sofort vom Sendetask unterbrochen, der die Queue wieder auffüllt und dann selbst in den "blocked State" geht. Dem Sendetask kann auch eine "block time" mitgegeben werden, in der er auf ein neues Ereignis wartet, bevor er wieder in den "blocked state" geht.

Empfangstask und Sendetask haben die gleiche Priorität:

Das bedeutet, dass der Sendetask die Queue füllt, bis er vom Scheduler unterbrochen wird oder die Queue voll ist. Erst dann geht er in den "blocked state". Der Empfangstask holt sich kontinuierlich Daten aus der Queue, bis sie leer ist, oder er vom Scheduler unterbrochen wird. Erst dann geht er in den "blocked state".

Mit der Funktion `taskYIELD()` kann jeder Task vorzeitig in den "blocked state" gehen.

3.5 Die FreeRTOS-Speicherverwaltung

FreeRTOS kennt drei Arten der Speicherverwaltung, um eine möglichst hohe Kompatibilität zu den unterschiedlichsten Systemen zu schaffen. Sie sind in den Dateien `heap_1.c`, `heap_2.c` und `heap_3.c` zu finden. Einzubinden ist also nur die Datei, die die gewünschte Verwaltung enthält. Bei allen drei Arten wird Speicher mit der Funktion `pvPortMalloc()` alloziert und mit `vPortFree()` freigegeben.

Schema 1 – heap_1.c

Schema 1 ist das einfachste von allen. Einmal allozierter Speicher kann nicht wieder freigegeben werden. Der deterministische [28] Algorithmus teilt ein einzelnes Array so oft auf, wie Anfragen an den Speicher gestellt werden. Dabei wird die maximale Array- bzw. Heapgröße durch die Definition `configTOTAL_HEAP_SIZE` begrenzt. Sie ist in der Datei FreeRTOSConfig.h (beschrieben unter Kapitel 3.9) zu finden. Diese Art der Verwaltung ist für Anwendungen gedacht, die zur Laufzeit keine Tasks oder Queues erstellen oder löschen, was für den WATOS-Player ausreichend ist.

Schema 2 – heap_2.c

Dieses Schema verwendet einen nicht-deterministischen „best fit“-Algorithmus und kann im Gegensatz zum Schema 1 Speicher wieder freigeben. Mehrere benachbarte, freie Blöcke können aber nicht zu einem großen, freien Block kombiniert werden. Das bedeutet, man sollte das Schema nicht verwenden, wenn oft Speicher alloziert und freigegeben wird, der unterschiedlich groß ist. Dies führt zu einer Speicherfragmentierung.

Im Schema 2 kann Speicher während der Laufzeit alloziert und freigegeben werden. Die maximale Heapgröße wird auch hier durch `configTOTAL_HEAP_SIZE` begrenzt.

Schema 3 – heap_3.c

Schema 3 ist ein Wrapper für die Standard-Speicherfunktionen `malloc()` und `free()`, die Thread-sicher gemacht wurden. Es ist unbedingt darauf zu achten, im Linker eine gültige maximale Größe für den Heap zu vergeben, wenn man dieses Schema verwendet.

3.6 Die FreeRTOS API

In Tabelle 3 befindet sich ein kleiner Ausschnitt an wichtigen API-Funktionen die auch bei der Programmentwicklung eine Rolle spielten. Eine genaue Beschreibung aller FreeRTOS-API Funktionen ist unter [11] zu finden.

Bereich	Funktion	Beschreibung
Task Creation	`xTaskCreate()`	Erstellt einen Task
	`vTaskDelete()`	Löscht einen Task
Task Control	`vTaskDelay()`	Task geht eine gewisse Zeit in den „blocked“ Mode relativ zum Aufruf von vTaskDelay()
	`vTaskDelayUntil()`	Task geht genau für eine gewisse Zeit in den „blocked“ Mode
Kernel Control	`taskENTER_CRITICAL()`	Betreten einer kritischen Sektion, Interrupts werden nicht beachtet
	`taskYIELD()`	Rechenzeit wird an andere Tasks abgegeben
	`taskEXIT_CRITICAL()`	Verlassen einer kritischen Sektion
	`vTaskStartScheduler()`	Starten des Scheduler
Queue	`xQueueCreate()`	Eine Queue wird erstellt
	`xQueueSend()`	An das Ende der Queue wird eine Nachricht gesendet
	`xQueueReceive()`	Eine Nachricht aus der Queue wird abgeholt
	`xQueueSendFromISR()`	Aus einer ISR heraus wird eine Nachricht in die Queue geschrieben
Synchronisation	`vSemaphoreCreateBinary()`	Eine Queue mit einem Eintrag wird erstellt um eine Ressource zu verwalten
	`vSemaphoreCreateCounting()`	Mehrere Ressourcen können verwaltet werden
	`xSemaphoreCreateMutex()`	Eine Queue mit einem Eintrag wird erstellt um eine Ressource zu verwalten
	`xSemaphoreTake()`	Mutex oder Semaphore wird belegt
	`xSemaphoreGive()`	Mutex oder Semaphore wird freigegeben
Co-Routine	`xCoRoutineCreate()`	Eine Co-Routine wird erstellt
	`vCoRoutineSchedule()`	Der Scheduler für die Co-Routine wird einmal ausgeführt
Speicher-verwaltung	`pvPortMalloc()`	Reserviert Speicher auf dem Heap
	`vPortFree()`	Gibt den reservierten Speicher wieder frei

Tabelle 3: FreeRTOS Schnittstellen API

3.6 Ein Minimales FreeRTOS-Beispielprogramm

Das Listing 3 zeigt ein einfaches FreeRTOS-Programm. In der Funktion `main` werden zwei Tasks mit den Argumenten: aufzurufende Funktion, Taskname, max. Stackgröße, Parameter, Priorität und Task-Handle erzeugt.

Der Parameter Task-Handle, wird nur dann gebraucht wenn ein Task einen anderen beeinflussen will. Dies geschieht über die Task-Handle-Referenz. Die anderen Argumente erklären sich von selbst; unter [11] sind alle Funktionen einschließlich Parameter beschrieben.

Die zwei Tasks geben jeweils einen String aus und warten dann eine gewisse Zeit (ticks), bevor sie wieder eine Ausgabe machen. Alles, was vor der `while(1)` Schleife steht, dient der Initialisierung

des Tasks und wird nur einmal aufgerufen, alles, was danach kommt, wird immer wieder vom Scheduler aufgerufen. Die Funktion `vApplicationIdleHook()` wird dann aufgerufen wenn kein anderer Task aktiv ist. Task1 wartet nach der Ausgabe 1000 ticks, also bei einer Tickrate von 2 ms wartet Task 1 zwei Sekunden. Danach wird Task2 aufgerufen. Er macht seine Ausgabe und wartet dann auch 2 Sekunden. So lange die zwei Tasks jetzt im Wartezustand sind (blocked), wird der Idle-Task ausgeführt.

```
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"


int i = 0; int a = 0; int y = 1;


   void TASK2(void* pvParameters)
  {
        while(1){
          i = !i;
          xil_printf("TEST");
          vTaskDelay(1000);
        }
   }
   void TASK1(void* pvParameters)
  {
        while(1){
        a = !a;
        xil_printf("hallo");
        vTaskDelay(1000);
        }
   }
   int main(void)
   {
        xTaskCreate(TASK2, "TASK2", 128, NULL, 1, NULL);
        xTaskCreate(TASK1, "TASK1", 128, NULL, 1, NULL);
        xil_printf("start\n");
        // Start Scheduler
     vTaskStartScheduler();
     return 0;
   }
```

```
      void vApplicationIdleHook( void )
      {
        y++;
      }
```

Listing 3: FreeRTOS minimales Beispielprogramm

3.6 Die Dateistruktur des FreeRTOS

Folgende Verzeichnisse und Dateien aus Listing 4 werden für die MicroBlaze-Portierung verwendet. Alle anderen Dateien können gelöscht werden.

```
FreeRTOS
 ¦
 +-Demo
 ¦   ¦
 ¦   +-Common              Demoanwendungen die für alle Ports gleich sind
 ¦   ¦   +-Minimal         Minimale Version der gemeinsamen Demodateien
 ¦   ¦   +-Full            Volle Version der gemeinsamen Demodateien
 ¦   ¦   +-include         Demo-Programm Header-Dateien
 ¦   ¦
 ¦   +-Microblaze          Demodaten speziell für MicroBlaze-Port
 ¦   ¦   +-serial
 ¦   ¦   +-ParTest
 ¦
 ¦
 +-Source                  Scheduler Quellcode
    ¦
     +-include             Scheduler Header-Dateien
     ¦
     +-portable            Scheduler Port-Layer für alle Ports
         ¦
         +-MemMang         Speicherverwaltung für alle Ports
         ¦
         +-GCC             Scheduler Port-Layer für GCC Compiler
            |
            ¦+-Microblaze  Scheduler Port-Dateien für GCC unter MicroBlaze
```

Listing 4: Verzeichnisstruktur FreeRTOS

3.7 Die FreeRTOS-Demoanwendung im Detail

Für jede unterstützte Platform gibt es eine Demoanwendung. Für das Spartan-3a-Board existiert leider keine. Jedoch lässt sich die Portierung für das Virtex-4-Board leicht anpassen.

- Es existieren 3 Flash-Tasks, die jeweils eine LED ansteuern und diese blinken lassen. Die erste LED hat die langsamste Frequenz, die letzte die schnellste.
- Für den Serial-Task gibt es jeweils für den Sende und den Empfangsvorgang eine LED. Sie blinken immer dann, wenn ein Datum empfangen oder gesendet wird.
- Da es nicht nur Tasks gibt, die eine visuelle Ausgabe haben, in der man sehen kann, ob der jeweilige Task funktioniert, gibt es einen Check-Task. Dieser überprüft, ob in den anderen Tasks Fehler auftreten.

  Der Check-Task überprüft alle 3 Sekunden alle anderen Tasks auf Fehler. Wird kein Fehler entdeckt, wechselt die LED den Zustand (an-aus). Wird aber ein Fehler entdeckt, blinkt diese LED mit einer Frequenz von 500 ms. Dieser Mechanismus kann überprüft werden, indem ein Loopback-Adapter an die serielle Schnittstelle gesteckt wird. Die LED wird alle 3 Sekunden blinken. Sobald der Adapter aber abgezogen wird, blinkt die LED alle 500 ms und zeigt dadurch einen Fehler an.

3.8 FreeRTOS für ein Zielsystem konfigurieren

FreeRTOS benötigt einen Timer, der den RTOS-Tick für den Prozess-Scheduler erzeugt. Außerdem wird ein Interrupt-Controller benötigt, weil nicht nur der Timer einen Interrupt erzeugt, sondern auch das selbst entworfene Soundmodul. In Abschnitt 9.1.2 befindet sich eine Anleitung, die beschreibt, wie man diese Hardwaremodule einbindet. Beim Einbinden von FreeRTOS in ein EDK10-Projekt gab es mehrere Probleme, die behoben werden mussten.

Vorgehen:

Alle nicht benötigten Daten aus dem Projektverzeichnis können gelöscht werden (siehe Abschnitt 3.6). Die restlichen Daten müssen in ein neues EDK10-Projekt eingebunden werden. Die Pfade der Header-Dateien müssen im Linker aktualisiert werden. Außerdem müssen alle Sektionen (Code, Heap, Stack) in den Hauptspeicher gelinkt werden. Die Datei `FreeRTOSConfig.h` muss richtig konfiguriert werden (siehe Abschnitt 3.3.1). Falls man die Datei `portmacro.h` in ein anderes als das ursprüngliche Verzeichnis verschoben hat, muss dieses in der Datei `portable.h` eingetragen werden. Schließlich muss man den EDK-9-Patch ausführen und die Compilerfehler ausbessern (siehe dazu Abschnitt 9.1.4 im Anhang).

3.9 Die Konfiguration über FreeRTOSconfig.h

Jedes Projekt benötigt noch die Header-Datei `FreeRTOSConfig.h`, die FreeRTOS konfiguriert. Die Einstellungen sind in der FreeRTOS Dokumentation erläutert [13]. Die wichtigsten Einstellungen und deren Beschreibung zeigt Tabelle 4.

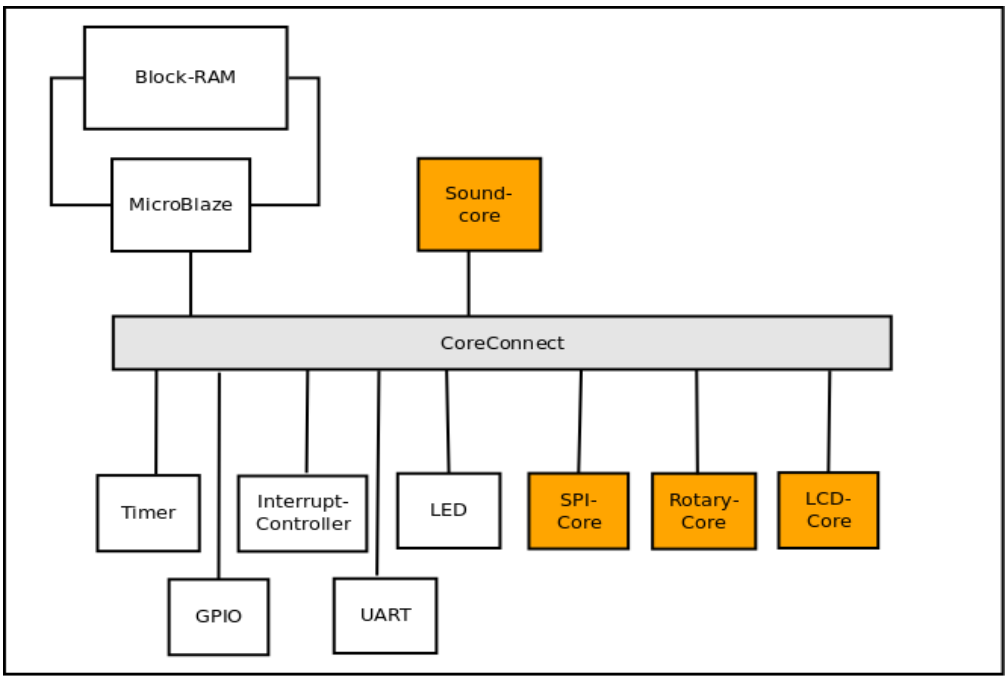
Makro	Wert	Beschreibung
configUSE_PREEMPTION	1	Für ein präemptives Multitasking
configUSE_IDLE_HOOK	0	Zum Aktivieren/Deaktivieren der Callback-Funktion Idle-Hook
configUSE_TICK_HOOK	0	Idle-Hook wird nach jedem Tick- Interrupt aufgerufen.
configCPU_CLOCK_HZ	50000000	CPU-Frequenz zur Berechnung der Timerperiode
configTICK_RATE_HZ	500	Frequenz des Tick-Interrupts für die Berechnung der Timerperiode
configMAX_PRIORITIES	4	höchste Priorität eines Tasks
configUSE_TRACE_FACILITY	1	Zur Beobachtung des Laufzeitverhaltens der Tasks
configMINIMAL_STACK_SIZE	120	Gibt die Stackgröße eines Task an. Es ergibt sich eine Größe von 480 Bytes bei einer Datenbreite von 4 Byte (120 x 4)
configTOTAL_HEAP_SIZE	18*1024	Gibt die Größe des Heap-Speichers an, die dem Kernel zur Verfügung steht. Es ergibt sich eine Größe von 73728 Bytes (4x16x1024)
configMAX_TASK_NAME_LEN	5	Maximallänge eines Strings für Tasknamen
configUSE_16_BIT_TICKS	0	Es wird ein 16 bit unsigned integer statt eines 32 bit unsigned integer für die Zeitmessung benützt
configUSE_CO_ROUTINES	0	Zur Aktivierung der Co-Routines.
configMAX_CO_ROUTINE_ PRIORITIES	2	Legt die höchste Priorität für eine Co-Routine fest.

Tabelle 4: Einstellungen in FreeRTOSConfig.h

4. Entwickelte Hardware-Module

Eigene Master- bzw. Slave-Module lassen sich im EDK sehr komfortabel durch einen Wizard einbinden. Dazu findet man im Internet jede Menge Informationen. Eine gute Anleitung ist unter [18] zu finden. Um eigene Module zu erstellen, muss man sich nicht explizit mit dem Core-Connect-Bus auseinandersetzen, dies macht das Intellectual Property Interface (IPIF). Der Wizard erstellt zwei Dateien, welche sich um die Buskommunikation kümmern. Die Top-Level-Datei hat den Projektnamen des Moduls. In die zweite Datei `user_logic.vhd` wird die User-Logik implementiert. Alle Module sind nach diesem Prinzip erstellt worden. Vorher musste jedoch ein funktionales SoC mit dem Wizard erstellt werden.

Einen möglichen internen Aufbau zeigt die Abbildung 11. Die orange hinterlegten Komponenten wurden selbst entworfen.



4.1 PCM - Master – 8

4.1.1 Überblick

Entwicklung:

Der Soundcore PCM Master 8 wurde in dem Wahlpflichtfach „Systems-on-a-Chip“ unter Herrn Prof. Dr. Kiefer von Florian Richter und dem Autor dieser Arbeit an der HS-Augsburg entwickelt.

Eine genaue Beschreibung über die Funktionsweise des Soundcores, Treiber und Anleitungen finden sich auf der Projektseite [19].

Abgrenzung:

In dieser Bachelorarbeit werden nur die wichtigsten Funktionen des Soundmoduls erklärt, weitere Informationen können durch den oben genannten Verweis bezogen werden.

Die Hardware wurde komplett aus dem SoC Praktikum übernommen, nur die Interruptfunktionalität wurde im Rahmen dieser Bachelorarbeit eingebaut, ebenso die Entwicklung der FreeRTOS tauglichen Treiber.

Funktion:

Der Soundcore kann Musikdaten im PCM-Rohformat abspielen (Wave-Files). Jedoch sind diese auf 11 kHz Samplerate und 8-Bit Auflösung beschränkt. Auch wird zur Zeit nur ein Kanal unterstützt. Der Soundcore holt sich seine Werte selbständig nach Initialisierung aus dem Speicher, er fungiert somit auch als DMA-Controller. Dazu benötigt er einen Masterzugang für den CoreConnect-Bus. Das bedeutet, er kann genauso wie die CPU Kontrolle über den Bus erlangen und diverse Adressierungen übernehmen. Dabei arbeitet der Soundcore mit zwei Adressbereichen, die abwechselnd ausgelesen werden. Immer dann, wenn der Soundcore einen Adressbereich abspielt, wird der andere von der Software mit Daten gefüllt. So wird eine unterbrechungsfreie Versorgung mit Sound-Daten gewährleistet. Da der Soundcore auch für die Bachelorarbeit "Linux auf einem FPGA" und für die vorliegende Arbeit verwendet wird, wurde er dahingehend optimiert, dass ein Ansprechen aus einem Betriebssystem heraus möglich ist. Der Soundcore wurde auf dem Xilinx Spartan-3E Development Kit in einem MicroBlaze SoC entworfen.

4.1.2 Hardware

Die Hardware des PCM-Master-8 teilt sich in zwei wichtige Bereiche.

Das wäre einmal das **PLB-Interface-Module**, durch das die Buskommunikation realisiert wird. Alle Informationen, die der Soundcore benötigt, werden in Slave-Registern gespeichert. Ist der Soundcore konfiguriert, lädt er selbstständig Daten aus dem Hauptspeicher. Dies wird mit der Masterlogik realisiert. Somit ist sowohl ein Master- als auch ein Slave-Interface für den PLB vorhanden. Abbildung 12 zeigt den Aufbau des gesamten Systems mit Busanbindung.

Der **Soundcore** in Abbildung 12 (grau hinterlegt) ist der zweite wichtige Bereich, er teilt sich wiederum in insgesamt vier Bereiche auf.

Die **Control-Unit** ist der Motor des Soundcores, sie übernimmt die komplette Organisation. Die

Unit benötigt 2 Adressen, an diesen erwartet sie eine bestimmte Anzahl an PCM 8 Bit Daten. Die Anzahl kann durch ein Slave Register vorgegeben werden, ebenso die Samplerate. Die Adressen werden nun abwechselnd geladen. Ist eine Adresse geladen, wird der Adressgenerator gestartet.

Der **Adressgenerator** inkrementiert die geladene Adresse mit dem Takt der Sample-Clock-Unit, bis der Adressbereich durchgelaufen ist. Die Adressen werden der Masterlogik zugeführt. Die Control-Unit lädt dann den nächsten Adressbereich und startet wieder den Adressgenerator und zwar so lange, bis der letzte Adressbereich geladen wurde. Dies muss durch die Software mitgeteilt werden.

Die **Sample-Clock-Unit** wird durch ein Slave-Register auf einen bestimmten Wert initialisiert. Dieser Wert wird mit dem Systemtakt dekrementiert. Wird die Null erreicht, taktet die Sample-Clock-Unit und der Vorgang beginnt erneut.

Bei jedem Takt bekommt die **Core-Unit** ein Datum von der Masterlogik. Dieses Datum wird dann mittels PWM Modulation an den Lautsprecher ausgegeben.

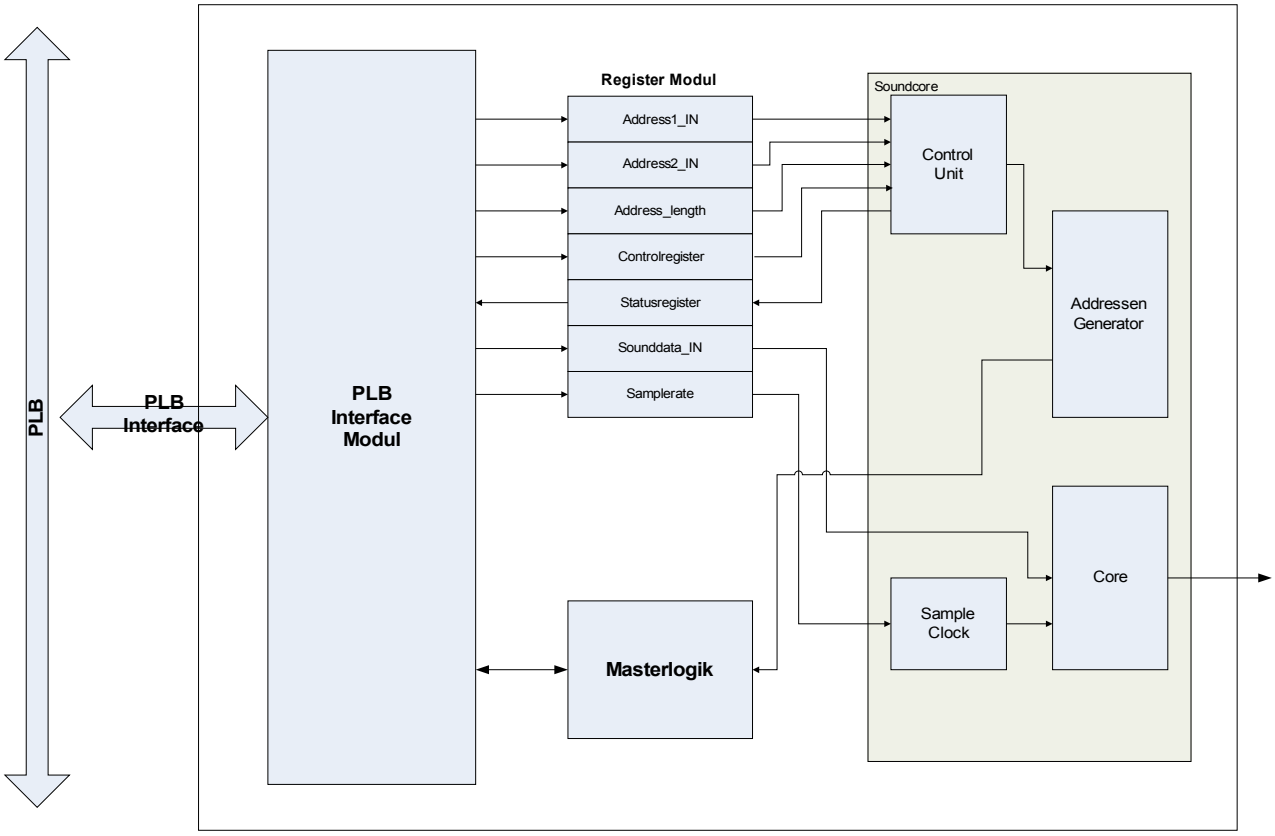


Abbildung 12: Aufbau Hardware Soundcore [3]

Das Core-Modul

Das Core-Modul erzeugt den eigentlichen Ton am Lautsprecher. Das Modul enthält einen Zähler, der kontinuierlich bis 256 (8 Bit) zählt. Er erzeugt somit eine PWM-Frequenz, die immer konstant ist. Bei 50 MHz Systemtakt des Spartan-3a Board werden somit 200 kHz erzeugt. Diese Frequenz ist deutlich außerhalb des hörbaren Bereichs und verursacht somit am Lautsprecher keinen Ausschlag. Die verschiedenen Tonpegel wiederum werden ebenfalls durch den Zähler erzeugt,

indem er zu einem bestimmten Zählerstand einen Ausgang auf 0 bzw. auf 1 schaltet. Einfach gesagt, wird nur das Pulspausenverhältnis geändert. Dadurch, dass man 256 verschieden Zählerstände anwählen kann, können auch 256 verschieden Pegel erzeugt werden. Zur Verdeutlichung der Funktionsweise findet sich im Listing 5 ein Quellcodebeispiel zur Erzeugung eines Tones. Die Variable `s_current_value_reg` enthält hierbei den Zählerstand.

```
process begin
      wait until rising_edge(clk);
      if (s_cnt<254) then s_cnt <= s_cnt+1;
      else               s_cnt <= 0;
      end if;
end process;

pwm_out_l <= '0' when (s_cnt>=s_current_value_reg and enable = '1')  else '1';
```

Listing 5: Implementierung Core-Modul

Die Soundcore Zustandsmaschine

Bei der in Abbildung 13 dargestellten Zustandsmaschine handelt es sich um einen Mealy-Automat. Die Soundcore Statemachine hat eigentlich nur die Aufgabe den Adressengenerator mit den richtigen Startadressen zu belegen und zum richtigem Zeitpunkt zu starten. Sie hat die folgende Funktionsweise:

Im Ausgangszustand befindet sich der Automat im Zustand `Idle`. Sobald das `enable`-Signal gesetzt wird, geht er in den Zustand `Load_Adr1`. Hier wird der Adressengenerator mit der gewünschten Startadresse und Länge geladen und das Signal `adr1_active` auf aktiv gesetzt. Einen Takt später springt der Automat schon in den nächsten Zustand und aktiviert den Adressengenerator. Der Adressengenerator generiert nun Adressen, wie oben beschrieben, und erzeugt schließlich bei Beendigung des Bereichs das Signal `addresses_done`. Das Gleiche wird nun für den zweiten Adressbereich durchgeführt. Zum `address_write`-Zustand muss noch Folgendes gesagt werden: Definiert man von außen mit dem `last_address`-Signal, dass es sich um den letzten Adressbereich handelt und der Adressengenerator signalisiert hat, dass er fertig ist, so wird in den Ausgangszustand zurückgesprungen. Es besteht die Möglichkeit, eine Interruptfunktionalität einzurichten und zwar genau dann, wenn die Signale `adr1_active` und `adr2_active` gesetzt werden.

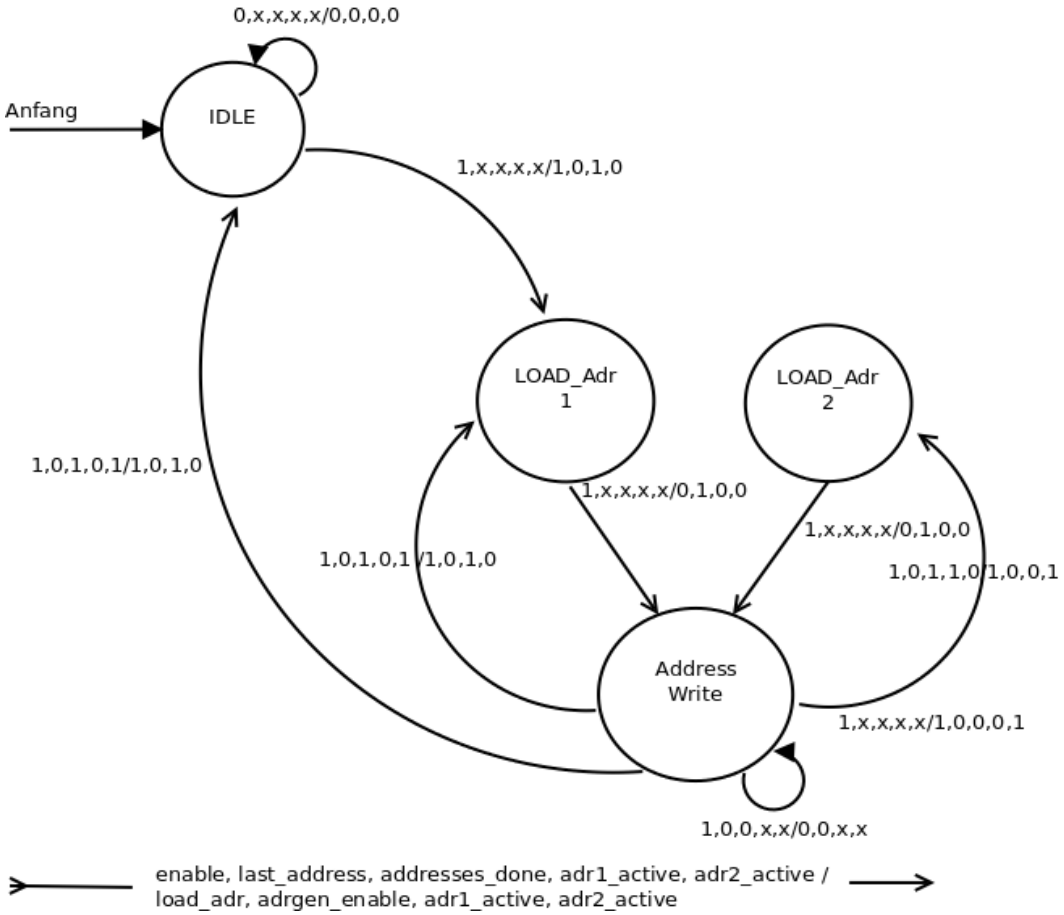


Abbildung 13: Soundcore Zustandsmaschine

Schnittstelle zwischen Soft- und Hardware

Als Schnittstelle zwischen Soft- und Hardware dienen insgesamt sechs Slave-Register. Über diese Register können alle nötigen Einstellungen, die zur Verwendung des Soundmoduls benötigt werden, angepasst werden. Tabelle 5 gibt einen Einblick in die vorhandenen Register. Tabelle 6 zeigt im Detail das Steuer- und Statusregister.

Im Steuerregister gibt das Signal `enable` den Startschuss für den Soundcore. Er beginnt zu arbeiten. Mit dem Signal `last_address` signalisiert man dem Soundcore, dass der letzte Adressbereich abgespielt wird. Danach endet die Soundausgabe automatisch. Das Signal `debug` wurde während der Entwicklung für die Fehleranalyse verwendet.

Das Statusregister zeigt stets an, welcher Adressbereich momentan bearbeitet wird. Dies geschieht durch die Signale `stat_address1` und `stat_address2`. `Debug` ist wiederum ein Signal, das nur während der Entwicklung eine Rolle spielte.

Bezeichnung	Offset	Beschreibung
Address1	0x0	Startadresse des ersten Adressbereichs
Address2	0x4	Startadresse des zweiten Adressbereichs
Length	0x8	Die Länge des aktuellen Adressbereichs
Statusregister	0xc	Statusregister
Steuerregister	0x10	Steuerregister
Samplerate	0x14	Einstellen der Samplerate

Tabelle 5: Alle Register des Soundcores

Bit	7	6	5	4	3	2	1	0
Steuerreg	NC	NC	NC	NC	NC	debug	enable	last_address
Statusreg	NC	NC	NC	NC	NC	debug	stat_address2	stat_address1

Tabelle 6: Status- u. Steuerregister im Detail

4.1.3 Interruptfunktionalität integrieren

Der Interrupt wird aktiviert, sobald sich die Statemachine in einem der beiden `CMD_LOAD` Zustände befindet. Im `CMD_ADDRESS_WRITE` Zustand wird der Interrupt wieder deaktiviert.

Die MHS-Datei des Soundcores muss um das neue Signal abgeändert werden:

```
PORT External_IRQ_0 = External_IRQ_0, DIR = I, SIGIS = INTERRUPT, SENSITIVITY =
EDGE_RISING
```

Listing 6: Erweiterung der MHS-Datei des Soundcores

Nun ist es nur noch nötig, die Signale im EDK richtig miteinander zu verschalten. Dies ist im Anhang unter Abschnitt 9.1.3 erklärt. Unter Abschnitt 9.1.7 sind alle notwendigen Softwareänderungen beschrieben, um den PCM-Master-8 am Interrupt-System anzumelden.

4.1.4 Software-Treiber

Die Software-Treiber sind in den Dateien `sound.c` und der zugehörigen Header-Datei `sound.h` zu finden.

Implementierte Funktionen:

Für den Betrieb des Soundcores werden die nachfolgend beschriebenen Funktionen und Strukturen benötigt.

Die folgende Struktur wird der Queue übergeben wird. Sie enthält die Sounddaten (PCM) und die Zeiger auf die Adresspuffer:

```
struct sound_config
{
 unsigned char *adr_puffer1;
 unsigned char *adr_puffer2;
 unsigned char buffer[PUFFER_SIZE];
};
```

Die Funktion `init_sound()` Initialisiert den Soundcore mit den Pufferadressen, der Pufferlänge und der Samplerate. Die Pufferlänge ist mit `PUFFER_SIZE` auf 512 Byte definiert, die Samplerate ist mit `PCM_MASTER_11_kHz` auf 11 kHz eingestellt:

```
void init_sound(  unsigned char *puf_adr_1,
                  unsigned char *puf_adr_2,
                  portBASE_TYPE puf_length,
                  portBASE_TYPE sample_rate)
```

Über die Funktion `void set_control_sound()` lässt sich der Soundcore steuern.
Es gibt die Argumente: `SOUND_ENABLE`, `SOUND_LAST_PUFFER` und `SOUND_DISABLE`.
Es können auch mehrere Argumente durch bitweises verbinden (logisches UND) übergeben werden:

```
void set_control_sound(unsigned portBASE_TYPE control_data)
```

Die folgende Funktion gibt an, welcher Puffer momentan bearbeitet wird. Sie kann mit den Argumenten `SOUND_STAT1`, `SOUND_STAT2` konfiguriert werden:

```
portBASE_TYPE get_status_sound(unsigned portBASE_TYPE status_data)
```

Die Funktion `fill_puffer_sound()` kopiert `data_legth` Bytes von `source_address` nach `puffer_address:`

```
void fill_puffer_sound( unsigned char *puffer_address,
                        unsigned char *source_address,
                        unsigned portBASE_TYPE data_length)
```

Die Funktion `manage_interrupt_sound()` kümmert sich um die Bearbeitung eines Sound-Interrupts. Sie ist dafür zuständig, dass der richtige Speicherbereich mit neuen Daten gefüllt wird. Als Parameter hat sie die oben besprochene Struktur; in ihr enthalten sind die Speicherbereiche der Puffer und die Pufferdaten:

```
void manage_interrupt_sound (struct sound_config *s_con1)
```

Die folgende Funktion schreibt Daten von der seriellen Schnittstelle an eine bestimmte Adresse. Als Rückgabe enthält sie die Anzahl der eingelesenen Bytes. Die Funktion war am Anfang, als die SD-Karte noch nicht funktionierte, die einzige Möglichkeit, große Daten in den Speicher zu schreiben. Wird in den Eingangsdaten 0x03 erkannt endet der Einlesevorgang. Der Wert kommt in PCM-Daten recht selten vor, deswegen wurde er ausgewählt:

```
unsigned int portBASE_TYPE fill_source_from_rs232(unsigned char
*source_address)
```

Port-Registrierung in der MPD-Datei

Um dem System mitzuteilen, welche Pins nach außen zu führen sind, muss die Datei `pcm_master8_v2_1.mpd` um folgende Zeilen erweitert werden. Außerdem wird hier dem System mitgeteilt, dass es sich bei dem Signal `interrupt_out` um ein Interrupt-Signal handelt.

```
PORT pwm_out_l = pwm_out_l, DIR = O, PERMIT = BASE_USER
PORT pwm_out_r = pwm_out_r, DIR = O, PERMIT = BASE_USER
PORT interrupt_out = "", DIR = O, SIGIS = INTERRUPT, SENSITIVITY =
LEVEL_HIGH, INTERRUPT_PRIORITY = HIGH
```

Tabelle 7: Registrieren der externen Ports für das Soundmodul

4.2 „Easy and Fast“-SPI-Core

4.2.1 Überblick

Um den Soundcore mit Daten zu versorgen, wurde als Speichermedium eine SD-Karte ausgewählt. SD-Karten sind sehr günstig, fast jeder Laptop oder PC hat schon im Auslieferungszustand ein Lese- bzw. Schreibgerät. SD-Karten sind bis zu einer Speichergröße von 32 GB zu haben. Sie werden mit einem Spannungspegel von 3,3 V betrieben, was kompatibel zu dem Spartan-3a-Chip ist. Ein weiterer Vorteil ist die einfache Ansteuerung über das SPI (Serial Peripheral Interface) Protokoll.

Um eine SD-Karte am Spartan-3a zu betreiben, benötigt man ein entsprechendes Lesegerät [29]. Abbildung 14 zeigt ein solches Gerät. Das Gerät hat nur die Aufgabe, die Pins der SD-Karte nach außen zu führen. Eine Spannungsstabilisierung und eine Betriebs-LED sind ebenfalls vorhanden. Das Gerät wird über den Erweiterungsstecker J19 oder J18 angeschlossen

SPI (Serial Peripheral Interface)

Das Serial Peripheral Interface ist ein von Motorola entwickeltes Bus-System. Digitale Schaltungen arbeiten nach dem Master-Slave-Prinzip und werden über eine `clk` Leitung synchronisiert. SPI benötigt 4 Leitungen: MOSI (Master out Slave in), MISO (Master in, SLAVE out), SS (Slave select) und CLK (Clock). Es gibt einen Master und beliebig viele Slaves. Jede Kommunikation geht vom Master aus. Der Master selektiert mit SS den Slave, den er ansprechen möchte. Dann gibt er den bis zu 25 MHz schnellen Takt auf die `clk` Leitung. Währenddessen sendet er einen Befehl über die MOSI Leitung. Ist der Befehl zu Ende, wird 0xFF gesendet und zwar so lange bis der Slave über die MISO Leitung geantwortet hat. Auch der Takt bleibt während dieser Zeit erhalten [30].

Implementierung

Um die SD Karte anzusteuern, wurde ein eigener SPI-Core entwickelt, da der XPS-SPI-Core nicht flexibel genug ist, um SD-Karten anzusprechen. Das erste Problem war, dass bei jedem Sendevorgang automatisch der eingestellte Slave selektiert wurde. Die SD-Ansteuerung erfordert aber jeweils am Anfang des zu sendenden Kommandos 8 Takte im inaktiven Zustand der SD-Karte.

Eine mögliche Lösung wäre es, einen Dummy-Slave einzurichten und für diesen Vorgang anzusprechen.

Das nächste Problem war, dass der Teiler mit dem Einbinden des XPS-SPI-Moduls eingestellt wird und später nicht mehr verändert werden kann. Das ist aber dringend nötig, da die SD-Karte mit max. 400 kHz initialisiert werden muss, und im Betrieb dann auf bis zu 25 MHz eingestellt werden kann.

Ein Lösung wäre, für den INIT-Vorgang eine langsamere Instanz von dem XPS-Modul einzurichten.

Aus diesen Gründen wurde ein einfaches und schnelles SPI-Slave-Modul entworfen. Es hat folgende Eigenschaften:

- SPI-Master-Modus
- 8 Bit Übertragungsmodus
- Variabler Takt einstellbar
- max. 1 Slave-Modul anschließbar

Das Modul kann jederzeit um weitere Funktionen ergänzt werden. Für die Ansteuerung der SD-Karte sind diese Funktionen jedoch ausreichend.

4.2.2 Hardware

Zustandsmaschine:

Die Zustandsmaschine in Abbildung 15, ein Mealy-Automat, startet im `STOP`-Zustand. Wird `enable = 1` erkannt, geht sie in den `Start`-Zustand über. In diesem Zustand werden Sende- u. Empfangsregister ausgelesen bzw. beschrieben. Wenn die Operation beendet ist, wird in den `IDLE`-Zustand gewechselt. Der Zustand ist wichtig, denn er hält die Hardware so lange an, bis die Software das Signal `enable` wieder auf 0 setzt. Fehlt er, gibt es Synchronisationsprobleme. Schließlich wird wieder in den `STOP`-Zustand gewechselt, wenn `enable` auf 0 geht.

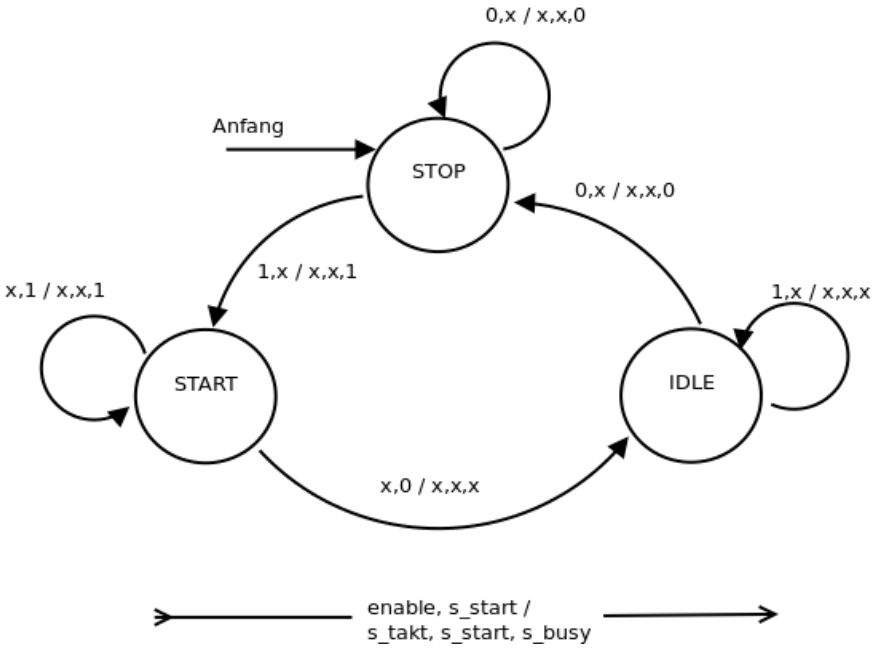
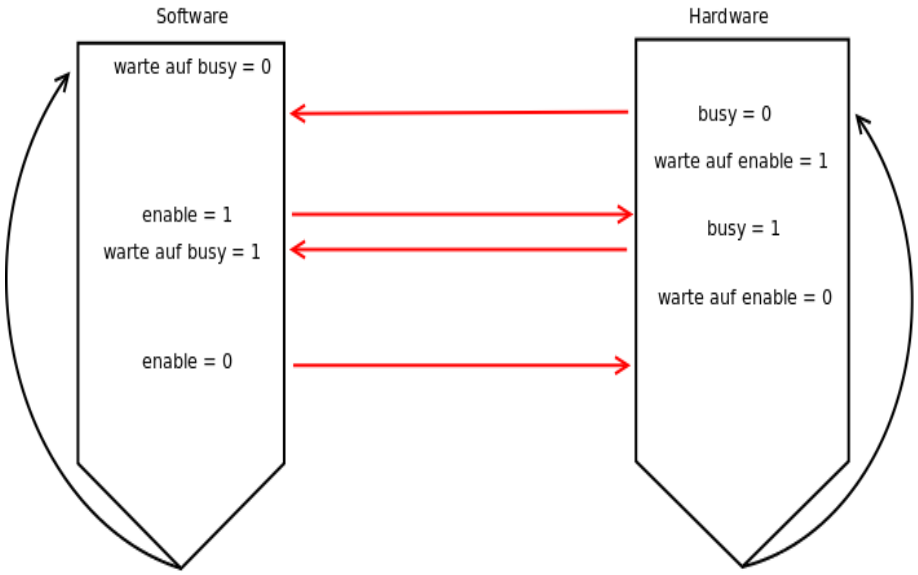


Abbildung 15: SPI-Modul-Zustandsmaschine

2-Wege-Handshake:

Um eine sichere Kommunikation zwischen Hardware und Software zu gewährleisten wurde ein 2-Wege-Handshake eingebaut. Ohne diesen Mechanismus kann es vorkommen, dass es zwischen Soft- und Hardware Synchronisationsprobleme gibt. Abbildung 16 zeigt den zeitlichen Ablauf.



Die Software wartet, bis das Signal `busy` Null ist, dann werden die Senderegister gefüllt und `enable` auf eins gesetzt. Erst dann verarbeitet die Hardware die Daten und signalisiert mit `busy` = 1, dass sie beschäftigt ist. Die Hardware pausiert dann so lange, bis die Software das Signal `enable` wieder auf Null setzt. Erst dann setzt die Hardware das Signal `busy` wieder auf 0. Mit diesem Vorgang sind beide Kommunikationspartner wieder synchronisiert.

Schnittstelle zwischen Soft- und Hardware

Als Schnittstelle zur Software dienen insgesamt 5 Register. In der Tabelle 8 ist eine Aufschlüsselung zu finden.

Steuerregister

Mit dem Signal `ss` wird das anzusprechende Gerät selektiert. Es kann nur ein Gerät angeschlossen werden. Wird `enable` auf Eins gesetzt, werden die Daten im Senderegister abgeschickt und ankommende Daten im Empfangsregister gespeichert. Es ist unbedingt darauf zu achten, dass vor dem Aktivieren von `enable`, der Teiler, sowie das Senderegister und das `ss`-Signal gesetzt wurden.

Statusregister

Sobald der SPI-Core Daten versendet oder empfängt, wird das `busy`-flag auf 1 gesetzt. So lange `busy` gesetzt ist, dürfen keine Informationen an den SPI-Core gesendet werden.

Empfangsregister

Hier stehen die ankommenden Daten. Um Daten zu erhalten müssen auch Daten gesendet werden. Es ist nur ein 8-Bit-Datum vorgesehen.

Senderegister

Hier stehen die zu versendenden Daten. Auch hier ist nur ein 8-Bit-Datum vorgesehen.

Teilerregister

Die Hardware ist so gebaut, dass ein maximaler Takt von 16,25 MHz mit einem Teiler von 1 erreicht wird. Dieser ist so gewählt, um eine maximale Kompatibilität für die verschiedenen SD-Karten zu gewährleisten.

Bit	7	6	5	4	3	2	1	0
Steuerregister	NC	NC	NC	NC	NC	NC	ss	enable
Statusregister	NC	NC	NC	NC	NC	NC	NC	busy
Empfangsregister	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
Senderegister	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
Teilerregister	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1

Tabelle 8: Register des SPI-Moduls

4.2.3 Simulation

Man hat die Möglichkeit entworfene Hardware in Xilinx-ISE zu simulieren. Dazu wird um die eigentliche Hardware eine Testbench geschrieben. Das bedeutet, dass alle Ein- und Ausgaben, die das zu testende Modul macht, in der Testbench generiert oder entgegengenommen werden. Der Hardware wird eine funktionale Umgebung vorgespielt. Dabei werden die Ein- und Ausgaben und interne Zustände der Hardware grafisch ausgegeben. Für jeden möglichen Testfall muss eine Auswertung gemacht werden. Die hier beschriebenen Simulationsergebnisse sind immer jeweils auf einen Testfall pro Hardwaremodul beschränkt.

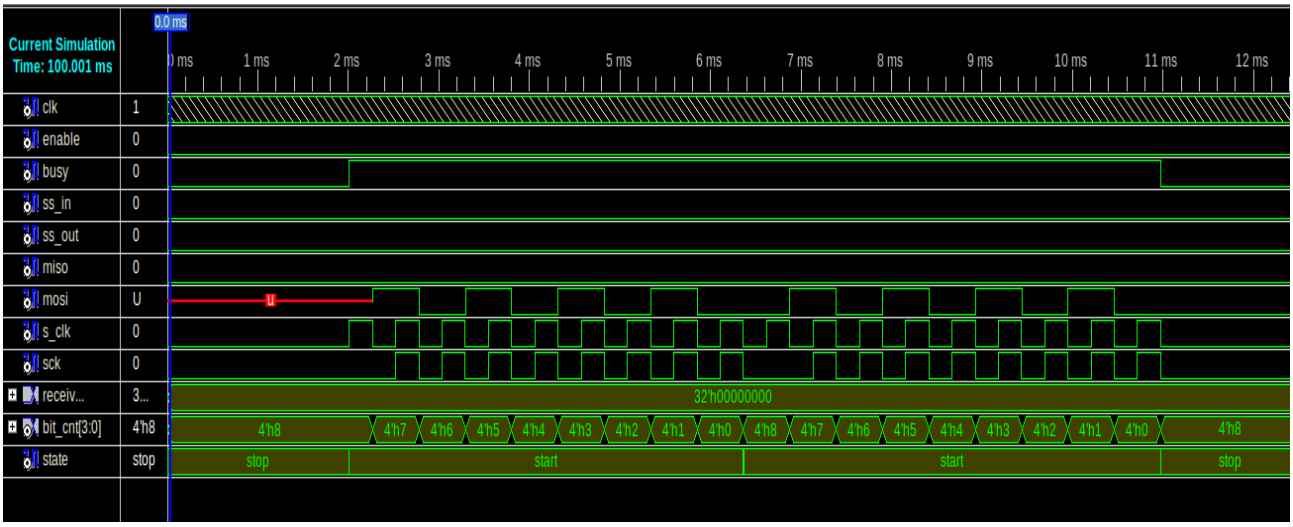
Listing 7 zeigt so einen Testfall. Er dient der Überprüfung des korrekten Sendeverhaltens. Dem SPI-Core wird eine funktionale Kommunikation vorgegaukelt. In der Testbench werden zuerst der Teiler und das Senderegister gefüllt. Dann erfolgt der oben besprochene Handshake. Die Testbench wartet, bis der SPI-Core bereit ist (`busy` = 0), dann wird `enable` auf 1 gesetzt. Erst wenn die Hardware `busy` wieder auf 0 setzt, ist die Kommunikation beendet und `enable` wird wieder auf 0 gesetzt.

```
wait for 1ms;
divider <= x"000000FF";       -- teiler 256

send_byte <= x"000000AA";
wait on busy until busy = '0';
enable  <= '1';
wait on busy until busy = '1';
enable  <= '0';

send_byte <= x"000000AA";
wait on busy until busy = '0';
enable  <= '1';
wait on busy until busy = '1';
enable  <= '0';
```

Listing 7: Ausschnitt aus der Testbench des SPI-Cores

Ergebnis der Simulation:

Abbildung 17 zeigt, dass das Datum `0xAA` zweimal hintereinander sauber getaktet wird. Das `busy`-Signal geht zwischen den Sendevorgängen kurz auf 0, ist aber wegen der Auflösung des Bildes nicht zu erkennen. Auch die Zustände der Zustandsmaschine sind abgebildet, der IDLE-Zustand ist ebenfalls nicht zu erkennen, weil er sehr kurz ist.



4.2.4 Software-Treiber

Die Software-Treiber sind in den Dateien `easy_spi.c` und der zugehörigen Headerdatei `easy_spi.h` zu finden.

Implementierte Funktionen:

Die folgende Funktion schreibt 0xFF ins Senderegister und liest dabei genau ein Byte vom Empfangsregister ein. Dabei wird der Handshakemechanismus benutzt. Wird ein weiteres Byte eingelesen, muss die Funktion erneut aufgerufen werden:

```
unsigned char mmc_read_byte(void)
```

Die Funktion `mmc_write_byte()` schreibt ein Byte in das Senderegister. Dabei wird der Handshakemechanismus benutzt. Werden weitere Bytes gesendet muss die Funktion erneut aufgerufen werden:

```
void mmc_write_byte(unsigned char byte)
```

Die anschließende Funktion schreibt ein Byte in das Teilerregister. Der Teiler kann jederzeit geändert werden. Das ist auch nötig, da der der Initialisierungsvorgang der SD-Karte mit max. 400 kHz getaktet werden darf. Danach kann der Takt erhöht werden:

```
void mmc_set_divider(unsigned int tmp)
```

Die Funktion `mmc_enable()` aktiviert den Slave-Select-Pin, um die SD-Karte aktiv zu schalten:

```
void mmc_enable(void)
```

Die Funktion `mmc_disable()` deaktiviert den Slave-Select-Pin, um die SD-Karte inaktiv zu schalten:

```
void mmc_disable(void)
```

Port-Registrierung in der MPD-Datei

Um dem System mitzuteilen, welche Pins nach außen zu führen sind, muss die Datei `lsd_spi_v2_1.mpd` um folgende Zeilen erweitert werden:

```
PORT mosi = mosi, DIR = O, PERMIT = BASE_USER
PORT miso = miso, DIR = I, PERMIT = BASE_USER
PORT sck = sck, DIR = O, PERMIT = BASE_USER
PORT ss_out = ss_out, DIR = O, PERMIT = BASE_USER
```

Listing 8: Registrieren der externen Ports für das SPI-Modul

4.3 Der „ENROT“ Dreh- / Druckknopf

4.3.1 Überblick

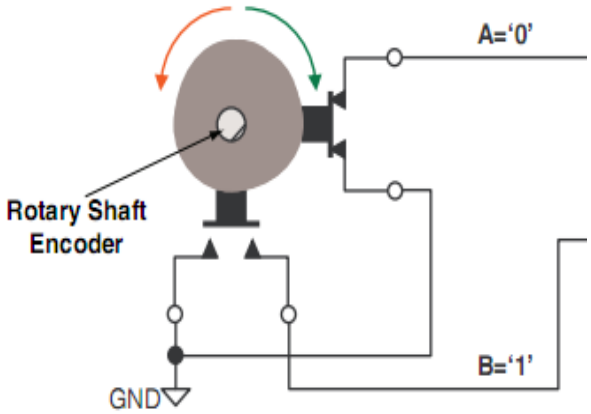
Auf dem Spartan-3a befindet sich ein Drehknopf umgeben von 4 Tastern, jeder an einer Himmelsrichtung positioniert. Der Drehknopf eignet sich hervorragend dafür, eine komfortable Menüführung zu realisieren. Er lässt sich um 360° drehen. Nach einer vollen Umdrehung wurde der Kontakt 20 mal geschlossen. Mit einer Drehung erreicht man somit 20 Einträge im Menü.

Der Drehknopf kann auch durch Drücken betätigt werden. Beide Formen der Eingabe müssen auf jeden Fall hardware- oder softwaremäßig entprellt werden. Der weitere Text beschreibt eine Entprellung durch nachgeschaltete Hardware.

4.3.2 Hardware des Dreh- / Druckknopf

In Abbildung 18 sieht man den physikalischen Aufbau des Drehknopfes. Er besteht aus zwei Kontakten, die auf Low gezogen werden. Dreht man jetzt den Knopf, öffnet ein Kontakt vor dem anderen, abhängig von der gewählten Richtung. Dreht man weiter, schließt ein Kontakt vor dem anderen. Wenn der Knopf im Ausgangszustand ist, sind beide Kontakte geschlossen. Zu beachten

ist, dass ein geschlossener Kontakt ein Low-Signal erzeugt.



Ein Taster wird nach dem Betätigen nicht sofort ein High- oder Low-Signal erzeugen, vielmehr wird er eine Kombination von diesen ausgeben, bevor er seinen endgültigen Zustand erreicht. Diesen Vorgang nennt man Prellen. Abbildung 19 zeigt ein solches Verhalten.

Abbildung 19: Der Drehknopf prellt [2]

Zustandsmaschine für den Drehvorgang:

Die Abbildung 20 zeigt die Zustandsmaschine als klassischen Moore-Automat in Kombination mit zwei Registern, in denen der Ausgangszustand gespeichert wird. In Verbindung mit der nachgeschalteten Entprellhardware wird aus dem Dreh- / Druckknopf der „ENROT"-Knopf. Nur im Zustand `s00`, `s11_l` und `s11_r` werden die Registerinhalte geändert, ansonsten bleiben sie erhalten. Die Zustandsmaschine erkennt ein Drehen des Drehknopfes und springt je nach Drehrichtung in den linken oder rechten Zweig. Wenn der Drehknopf prellt, wird in den vorherigen Zustand zurückgesprungen, und dann wieder nach vorne. Das passiert beim Prellen am Anfang sowie am Ende. Im Zustand `s_11_l` und `s_11_r` wird das jeweilige Register auf 1 geschaltet. Erst im `s_00`-Zustand werden die Register wieder auf 0 gesetzt.

Beispiel: Drehen nach rechts

`A` (siehe Abbildung 19) geht von 0 auf 1, daraufhin geht die Zustandsmaschine in den Zustand `s_10_r`. `A` prellt und geht wieder auf 0. Die Zustandsmaschine geht zurück in den `s_00`-Zustand. `A` geht wieder auf 1 , die Zustandsmaschine geht wieder in den `s_10_r`-Zustand. Das passiert jetzt so oft, wie der Drehknopf prellt. `B` geht auf 1. Der Zustand der Zustandsmaschine wechselt auf `s_11_r`. Jetzt wurde eine Rechtsdrehung erkannt, und der Ausgang `reg_r` geht auf

1. `B` prellt und geht wieder auf 0. Die Zustandsmaschine springt einen Zustand zurück. `B` geht wieder auf 1. Die Zustandsmaschine ist wieder im Zustand `s_11_r`. Das passiert jetzt so oft, wie der Drehknopf prellt. `A` geht wieder auf 0. Der Zustand ändert sich auf `s_01_r`. Das Prellen wird wieder durch Rück- bzw. Vorspringen kompensiert. `B` geht auf 0. Der Zustand wechselt auf `s_00`. Der Ausgang `reg_r` geht auf 0, da der Drehvorgang beendet wurde. Das letzte Prellen verursacht einen Wechsel in den `s_01_l`-Zustand, aber nur, bis es vorbei ist. Der letzte Zustand ist dann wieder `s_00`.

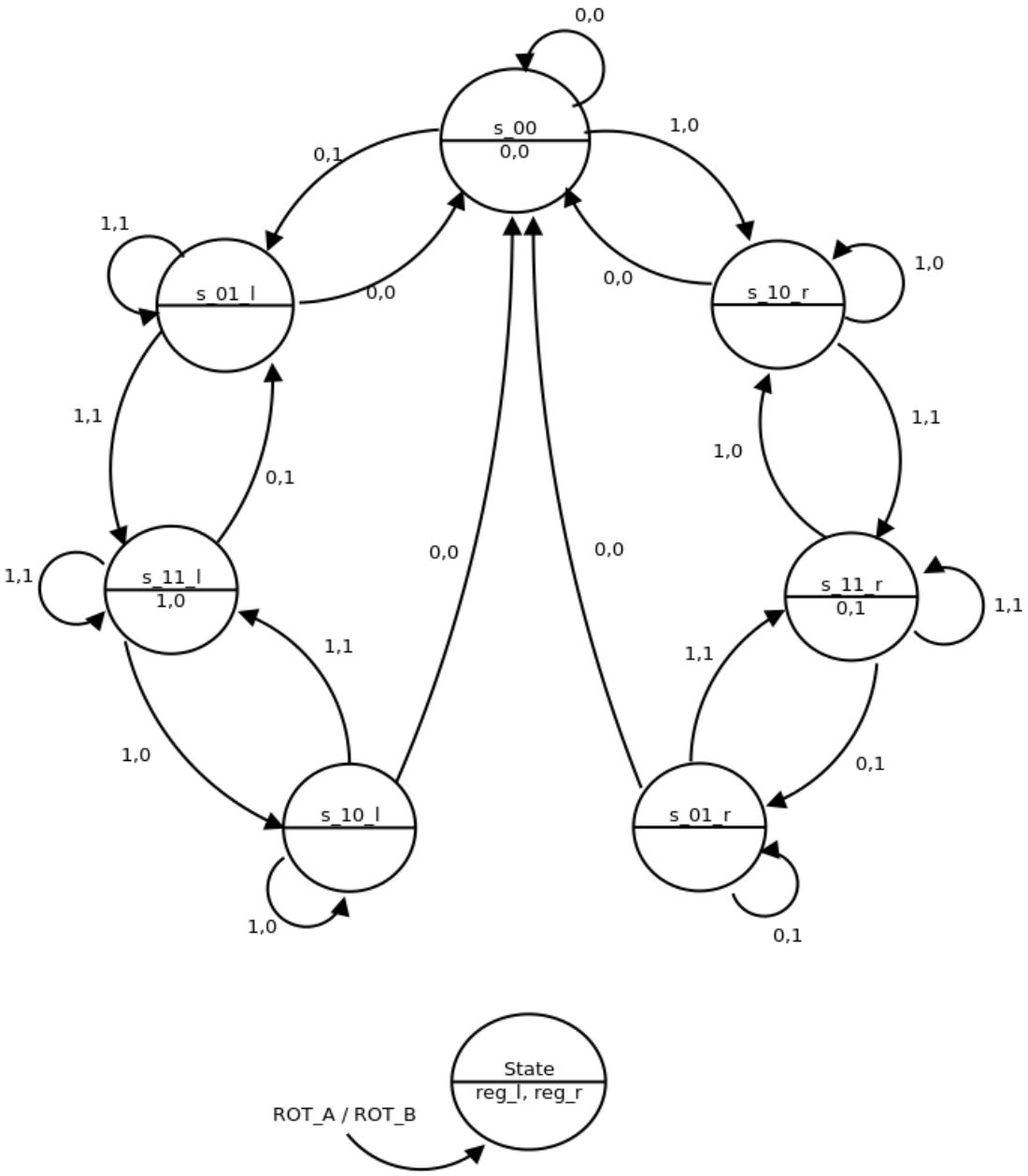


Abbildung 20: Zustandsmaschine für den Drehknopf

Entprellung des Druckknopfes:

Die Entprellung des Druckknopfes ist sehr einfach zu erreichen. Listing 9 zeigt die VDHL Implementierung. Sobald erkannt wurde dass der Button einmal eine 1 geliefert hat, läuft ein Zähler (`count`) von Null auf einen bestimmten Wert . Hat er den Wert erreicht ohne dass `ROT_CENTER` wieder 0 wird, ist das Prellen vorbei und der Ausgang wird auf 1 geschaltet, ansonsten wird der Zähler erneut anfangen zu zählen.

Den richtigen Wert für den Zählerstand gilt es durch Probieren einzustellen. Angenommen, ein Taster prellt mit ca. 20 kHz (das sind 50us), dann geht er nach ca. 25us wieder auf 0. Der Zähler sollte diese Zeit also überbrücken, damit er die Null, die durch das Prellen entsteht, erkennt. Der Zähler müsste somit 50us lang inkrementieren, dass wären bei einer Frequenz von 50 MHz ca. 2500 Werte.

```
PUSH: process (clk)
begin
if clk'event and clk = '1' then

  if (ROT_CENTER=tmp_key) then
    count <= 0;
  else
    count <= count+1;
  end if;

  if (count=2500) then
    tmp_key <= ROT_CENTER;
  end if;
end if;
end process;
```

Listing 9: Ausschnitt des VHDL Codes zur Entprellung des Druckknopfes

Schnittstelle zwischen Soft- und Hardware

Da keine Konfigurationsmöglichkeiten für „ENROT“ existieren, gibt es auch nur ein Statusregister. Dieses gibt die aktuellen Ereignisse am Button zurück. Bei jedem Ereignis wird ein Interrupt ausgelöst, was durch das Interrupt-Flag signalisiert wird. „ENROT“ ist aber nicht am Interrupt-System angemeldet. Er wird durch Pollen abgefragt. Tabelle 9 zeigt den Inhalt des Statusregister.

Bit	7	6	5	4	3	2	1	0
Statusregister	NC	NC	NC	NC	interrupt	center	left	right

Tabelle 9: Statusregister „ENROT“

4.3.3 Simulation des „ENROT“-Druckknopfes

In der Testbench aus Listing 10 wird der „ENROT“-Druckknopf, einschließlich seines Prellverhaltens simuliert. Bevor endgültig eine 1 an der Hardware aus Listing 10 anliegt wechselt der Zustand zweimal. Dies geschieht beim Drücken und beim Loslassen.

```
ROT_CENTER <= '1';                  ROT_CENTER <= '0';
wait for 1us;                       wait for 1us;
ROT_CENTER <= '0';                  ROT_CENTER <= '1';
wait for 1us;                       wait for 1us;
ROT_CENTER <= '1';                  ROT_CENTER <= '0';
wait for 1us;                       wait for 1us;
ROT_CENTER <= '0';                  ROT_CENTER <= '1';
wait for 1us;                       wait for 1us;
ROT_CENTER <= '1';                  ROT_CENTER <= '0';
wait for 100us;                     wait for 10us;
```

Listing 10: Ausschnitt aus der Testbench des „ENROT“-Druckknopfes

Ergebnis der Simulation:

Das Ergebnis in Abbildung 21 zeigt, dass der Knopf erfolgreich entprellt wurde. Auffallend ist die Verschiebung zwischen `rot_center` und `center`. Das liegt an der Methode des Entprellens durch das Inkrementieren eines Registers auf einen bestimmten Wert.

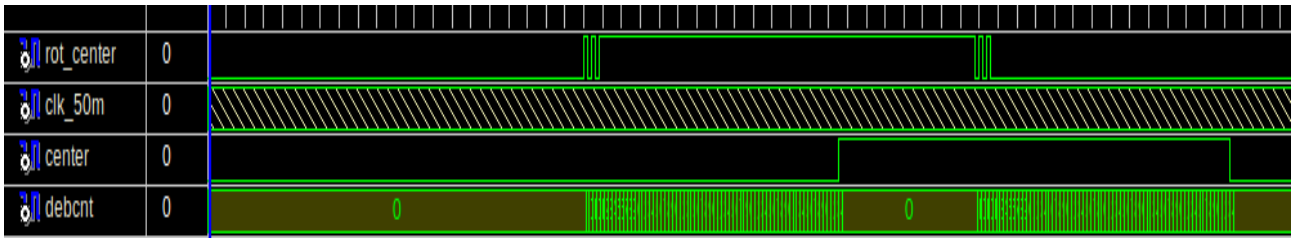


Abbildung 21: Simulationsergebnis des „ENROT“-Druckknopfes

4.3.4 Simulation des „ENROT“-Drehknopfes

Die Testbench des „ENROT“-Drehknopfes unterscheidet sich nicht wesentlich von der des „ENROT“-Druckknopfes. Im Prinzip muss nur ein Signal mehr entprellt werden. Somit ist das erzeugte Signal von der Kombination zweier Eingangssignale abhängig. Listing 11 zeigt einen Ausschnitt aus der Testbench. Beide Signale wechseln zwei Mal den Zustand, bevor sie endgültig

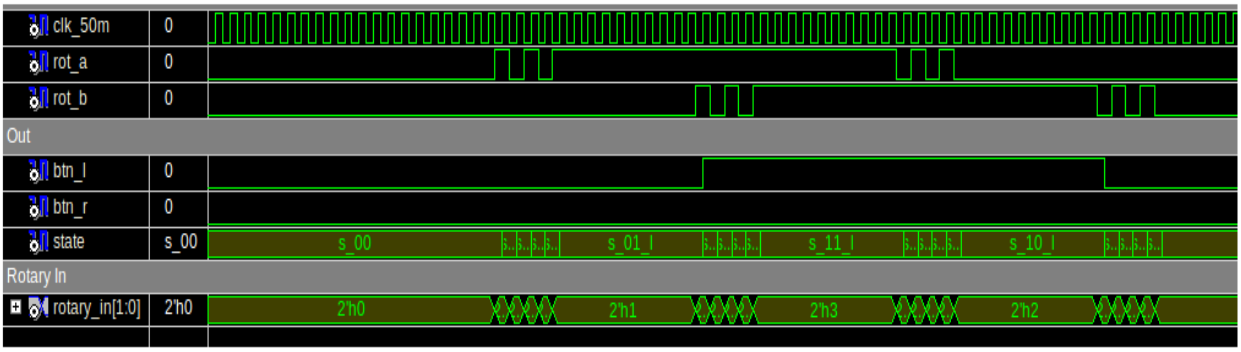
eine Eins liefern. Das Signal ROT_B ist zum Signal ROT_A zeitlich versetzt um eine Rechtsdrehung zu simulieren.

```
--Prellen Signal A am Anfang          --Prellen Signal A am Schluss
ROT_A <= '1';.                        ROT_A <= '0';
wait for 1us;                         wait for 1us;
ROT_A <= '0';                         ROT_A <= '1';
wait for 1us;                         wait for 1us;
ROT_A <= '1';                         ROT_A <= '0';
wait for 1us;                         wait for 1us;
ROT_A <= '0';                         ROT_A <= '1';
wait for 1us;                         wait for 1us;
ROT_A <= '1';                         ROT_A <= '0';
wait for 10us;                        wait for 10us;
--Prellen Signal B am Anfang          --Prellen Signal B am Schluss
ROT_B <= '1';                         ROT_B <= '0';
wait for 1us;                         wait for 1us;
ROT_B <= '0';                         ROT_B <= '1';
wait for 1us;                         wait for 1us;
ROT_B <= '1';                         ROT_B <= '0';
wait for 1us;                         wait for 1us;
ROT_B <= '0';                         ROT_B <= '1';
wait for 1us;                         wait for 1us;
ROT_B <= '1';                         ROT_B <= '0';
wait for 10us;
```

Listing 11: Ausschnitt aus der Testbench des „ENROT“-Drehknopfes

Ergebnis der Simulation

Das Simulationsergebnis in Abbildung 22 zeigt, dass der Drehknopf sauber entprellt wurde. Im Gegensatz zur Entprellung des Druckknopfes gibt es hier keine Verschiebung des Ausgangssignales zum Eingang, da jedes Prellen genau registriert wird. Sobald klar ist, in welche Richtung gedreht wurde, wird sofort das entsprechende Register gesetzt. Eine Linksdrehung oder ein Rechts-/Linkswechsel wurden nicht explizit simuliert, aber durch einen Praxis-Test erfolgreich überprüft.



4.3.5 Software-Treiber für „ENROT“

Die Software-Treiber sind in den Dateien `rotary.c` und der zugehörigen Headerdatei `rotary.h` zu finden.

Implementierte Funktionen:

Die folgende Funktion gibt die aktuelle Drehrichtung des Drehknopfes zurück. Wird jedoch eine Drehung nicht vollständig ausgeführt, wird immer die Drehrichtung zurückgegeben, bis der Drehknopf wieder in seinem Ausgangszustand ist. Im Zuge des Zählvorgangs im Hauptprogramm wird dieses Problem mit der Funktion `get_rotary_count()` behoben. Jede Drehung wird nur einmal gezählt, auch wenn sie nicht vollständig ausgeführt wurde:

```
char get_rotary_direction(void)
```

Die anschließende Funktion gibt eine 1 zurück so lange der Button gedrückt wird. Wird der Button losgelassen wird eine 0 zurückgegeben:

```
char get_button_center(void)
```

Die Funktion `get_button_center_once()` gibt einen 1 Impuls zurück, wenn der Button gedrückt wird. Für eine erneute Rückgabe muss der Knopf losgelassen und wieder gedrückt werden:

```
char get_button_center_once(void)
```

Port-Registrierung in der MPD-Datei

Um dem System mitzuteilen, welche Pins nach außen zu führen sind, muss die Datei `rotary_button_v2_1_0.mpd` um folgende Zeilen erweitert werden.

```
PORT ROT_A = ROT_A, DIR = I, PERMIT = BASE_USER
PORT ROT_B = ROT_B, DIR = I, PERMIT = BASE_USER
PORT ROT_CENTER = ROT_CENTER, DIR = I, PERMIT = BASE_USER
```

Listing 12: Registrieren der externen Ports für den Druck- / Drehknopf

4.3 LC-Display

4.3.1 Überblick

Auf dem Spartan-3A-Board befindet sich der integrierte Schaltkreis Hitachi-HD44780. Er ist ein Standard-Industrie-Controller für kleine alphanumerische Dot-Matrix-LC-Displays. Er übernimmt die komplette Ansteuerung inklusive Erzeugung aller benötigten Signale. Die Darstellung von Text wird durch einen integrierten Zeichengenerator erleichtert.

Solche Anzeigemodule sind in den Konfigurationen 1 x 8 Zeichen bis 40 x 4 Zeichen verfügbar. Sie enthalten den HD44780 und, falls erforderlich, den Spaltentreiber HD44100 bereits auf dem Modul integriert. Das verwendete Display kann 2 x 16 Zeichen darstellen.

Das Anzeigemodul ist somit bereits anschlussfertig für die Verwendung in diversen Mikrocontroller-Schaltungen. Es gibt 2 mögliche Ansteuerungen; entweder der 4-Bit-Modus oder der 8-Bit-Modus. Beim 4-Bit-Modus muss das Byte halbiert werden, es wird zuerst das High-Nibble übertragen dann das Low-Nibble. Das LCD wird im 4-Bit-Modus angesprochen, da auch nur 4 Datenleitungen mit dem FPGA verbunden sind.

Ansteuerung:

Zuerst muss eine Initialisierung durchgeführt werden. Hier wird dem Controller mitgeteilt, wie groß das Display ist, außerdem der Modus der Datenübertragung (entweder 4-Bit oder 8-Bit), Position des Cursors etc..

Die Initialisierung benötigt ein genaues Timing, die genaue Ansteuerung ist unter [5] erklärt.

An das Display werden entweder Befehle oder Daten gesendet. Unterschieden wird dies durch das RS-Signal. Bevor ein Datum angelegt wird, muss also angegeben werden, ob ein Datum oder ein Steuerbefehl kommt. Ein Steuerbefehl kann z.B. den Cursor positionieren. Dann wird das E-Signal auf 1 gesetzt, 1 us gewartet und dann wieder auf 0 gesetzt. Jetzt ist das Datum übernommen worden.

Wird der 4-Bit-Modus verwendet, muss dieser Vorgang je für High- und Low-Nibble durchgeführt werden. Das Display interpretiert das empfangene Byte als ASCII-Code [6].

Beschaltung:

Tabelle 10 zeigt das Anschlussschema des LC-Displays. Das LCD ist am Spartan-3a im 4-Bit Modus angeschlossen. Das Display hat keine Hintergrundbeleuchtung, daher sind die Pins 15 und 16 nicht angeschlossen.

Pin	Funktion	Beschreibung
1	VSS	GND
2	VCC	Versorgungsspannung 5V
3	V0	Kontrastspannung
4	RS	Registerauswahl 0 = Befehl 1= Datum
5	R / W	Schreiben / Lesen
6	E	Taktleitung / Enable
7	DB0	Datenleitung (8 Bit Modus) n.c.
8	DB1	Datenleitung (8 Bit Modus) n.c.
9	DB2	Datenleitung (8 Bit Modus) n.c.
10	DB3	Datenleitung (8 Bit Modus) n.c.
11	DB4	Datenleitung
12	DB5	Datenleitung
13	DB6	Datenleitung
14	DB7	Datenleitung
15	A	Anode Hintergrundbeleuchtung n.c.
16	K	Kathode Hintergrundbeleuchtung n.c.

Tabelle 10: Beschaltung des LC-Displays

4.3.2 Hardware

Da die Ansteuerung des Displays komplett im Software-Treiber implementiert wurde ist nur ein sehr einfaches Hardwaremodul nötig. Es werden nur die benötigten Pins des Display auf Slave-Registern abgebildet, dies geschieht in der `user_logic.vhd`. Tabelle 11 zeigt einen Code-Ausschnitt dieser Datei.

```
LCD_D8          <= slv_reg0(31);
LCD_D9          <= slv_reg0(30);
LCD_D10         <= slv_reg0(29);
LCD_D11         <= slv_reg0(28);
LCD_RS          <= slv_reg0(27);
LCD_E           <= slv_reg0(25);
LCD_RW          <= '0';
```

Tabelle 11: VHDL Code Display Hardware-Modul

Schnittstelle zwischen Soft- und Hardware

Da vom Display nichts gelesen wird, ist auch kein Statusregister notwendig. Durch das Steuerregister (Listing 13) werden alle wichtigen Eingänge des LCD abgebildet. Bit 0-3 ist das Daten-Nibble. Bit 4 zeigt an, ob ein Befehl(0) oder ein Datum(1) am Daten-Nibble anliegt. Bit 5 zeigt an, ob gelesen oder geschrieben wird. Dieses Signal könnte man auch einfach mit Masse verbinden. Gibt man einen 1 Impuls auf Bit 6, wird das Daten-Nibble übernommen.

Bit	7	6	5	4	3	2	1	0
Steuerregister	NC	E	R/W	RS	DB7	DB6	DB5	DB4

Listing 13: Steuerregister des LC-Displays

4.3.3 Software-Treiber für das LC - Display

Die Software-Treiber sind in den Dateien `lcd.c`, `time.c` und in den zugehörigen Headerdateien `lcd.h` und `time.h` zu finden.

Implementierte Funktionen:

Die folgende Funktion kümmert sich um die Verzögerung. Die übergebene Variable enthält die Zeit in ms, um die verzögert werden soll. Die Funktion liest den aktuellen Timer-Wert aus und pollt ihn so lange, bis der aufaddierte Wert erreicht wird. Verursacht das Aufaddieren einen größeren Wert als den maximal durch den Timer festgelegten, wird dies korrekt berücksichtigt. Der Timer wird dabei nicht verändert und kann somit auch noch für andere Aufgaben verwendet werden. Die Funktion ist in der Datei `timer.c` zu finden:

```
void _delay_ms(int zeit)
```

Die Funktion `lcd_data()` sendet ein Datum an das LCD. Dabei wird zuerst das H- und dann das L-Nibble gesendet. Das zu sendende Datum wird als ASCII-Wert interpretiert. Die Funktion befindet sich in der Datei `lcd.c`:

```
void lcd_data(unsigned char temp1)
```

Die anschließende Funktion sendet ein Kommando an das LCD. Die Funktion befindet sich in der Datei `lcd.c`:

```
void lcd_command(u8 temp1)
```

Die wichtigsten Kommandos für das LC-Display sind in der Tabelle 12 beschrieben.

Befehl	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Beschreibung
del	0	0	0	0	0	0	0	0	0	0	löscht Display, Curser auf Adresse 0
start	0	0	0	0	0	0	0	0	0	*	Curser auf Adresse 0
entry	0	0	0	0	0	0	0	0	I/D	S	I/D Curser Laufrichtung, S: Shift an/aus
on/off	0	0	0	0	0	0	1	D	C	B	D: Display an/aus, C: Curser an/aus, B: Curser blinken
curser	0	0	0	0	0	1	S/C	R/L	*	*	S/C: Display o. Curser , R/L: nach rechts o. links
func.	0	0	0	0	1	DL	N	F	*	*	DL: 8/4Bit, N: 1/2 zeilig, F:5x8/5x11 Darstellung

Tabelle 12: Steuerbefehle für das LC-Display

Die Funktion `lcd_init()` initialisiert das Display mit den oben beschrieben Kommandos. Die Funktion befindet sich in der Datei `lcd.c`:

```
void lcd_init(void)
```

Port-Registrierung in der MPD-Datei

Um dem System mitzuteilen welche Pins nach außen zu führen sind, muss die Datei `lcd_display_v2_1.mpd` um folgende Zeilen erweitert werden.

```
PORT LCD_D9 = LCD_D9, DIR = O, PERMIT = BASE_USER
PORT LCD_D10 = LCD_D10, DIR = O, PERMIT = BASE_USER
PORT LCD_D11 = LCD_D11, DIR = O, PERMIT = BASE_USER
PORT LCD_D8 = LCD_D8, DIR = O, PERMIT = BASE_USER
PORT LCD_E = LCD_E, DIR = O, PERMIT = BASE_USER
PORT LCD_RS = LCD_RS, DIR = O, PERMIT = BASE_USER
PORT LCD_RW = LCD_RW, DIR = O, PERMIT = BASE_USER
```

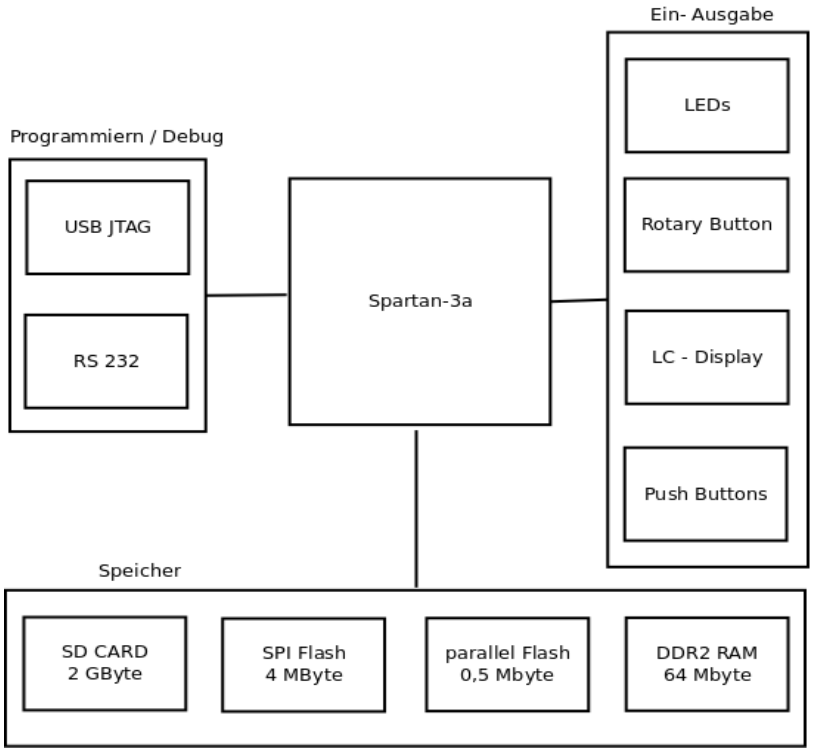
Tabelle 13: Registrieren der externen Ports für das LCD-Modul

5. Der „WATOS“-Player

5.1 Überblick

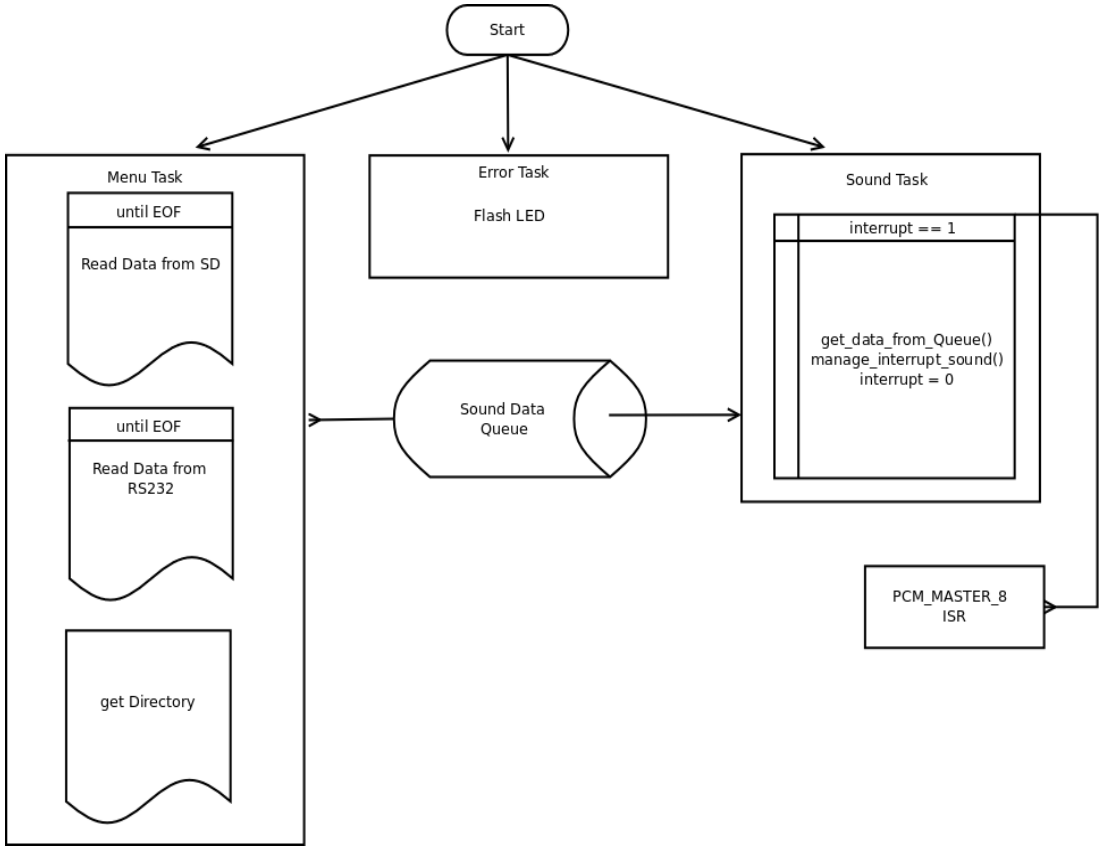
Der „WATOS“-Player (WAVE-RTOS-Player) liest WAVE-Dateien von der SD-Karte ein und leitet diese an den Soundcore weiter. Dargestellt werden diese Dateien auf dem LC-Display des Spartan-3a. Unter Verwendung des Drehknopfes und der Druckknöpfe kann ein Lied ausgesucht und abgespielt werden.

Um den Player erfolgreich zu betreiben sind mehrere externe Komponenten nötig. Abbildung 23 zeigt alle nötigen Komponenten. Für die Benutzerinteraktion steht ein Ein-/Ausgabeblock zur Verfügung. Mit einer LED wird der Status des ganzen Systems angezeigt. Das LCD gibt die auf der SD-Karte befindlichen Titel aus. Eingaben tätigt man entweder durch den Drehknopf, oder durch die Druckknöpfe. Als Speichermedium dient im Block-Speicher eine SD Karte mit 2GB-Kapazität. Als Hauptspeicher kommt der auf dem Spartan-3a-Board befindliche DDR2-RAM zum Einsatz. Der SPI-Flash und der parallele Flash Speicher sind nur dann notwendig, wenn man WATOS autonom, ohne PC, betreiben will. Der Debug- / Programmierblock ist notwendig, um das System zu initialisieren und um Debug-Mitteilungen auszulesen. Außerdem besteht die Möglichkeit, via RS232 Sounddaten direkt in den DDR2-RAM zu übertragen.



5.2 Hauptprogramm

Das Hauptprogramm verwendet insgesamt 3 Tasks und eine Queue. Alle drei Tasks haben die gleiche Priorität. Daneben existiert noch eine globale `Init`-Funktion, in der alle benutzten Geräte initialisiert werden. In der Hauptfunktion werden die Tasks und die Queue instantiiert. Bei der Gestaltung des Hauptprogrammes stand der konkurrierende Zugriff auf die gemeinsamen Ressourcen im Mittelpunkt. Deswegen wurden die vorhanden Ressourcen auf den exklusiven Zugriff bestimmter Tasks aufgeteilt. Auf die SD-Karte, sowie auf Ein- und Ausgabegeräte hat nur der Menü-Task Zugriff. Der Menü-Task verwaltet somit einerseits die Benutzerschnittstelle, andererseits ist er für das Einlesen der PCM-Rohdaten zuständig. Das Soundmodul steht exklusiv dem Sound-Task zur Verfügung. Der Sound-Task schreibt die konkreten PCM-Werte in einen vorher definierten Adressbereich des Soundcores. Für die Soundausgabe standen zwei Quellen zur Verfügung. Es können Sounddaten über die RS232-Schnittstelle und über die SD-Karte empfangen werden. Um zu vermeiden, dass zwei Quellen ggf. gleichzeitig auf den Soundcore zugreifen, wurde eine Queue verwendet. Der Soundcore bekommt die konkreten WAVE-Daten nur durch die Queue, und nur diese Daten werden abgespielt. Dabei spielt es jetzt keine Rolle mehr durch welchen Task oder wann die Daten kommen, da die Queue gegen konkurrierende Zugriffe geschützt ist. Abbildung 24 zeigt den Programmaufbau.



5.2.1 Menu-Task

Der Menu-Task ist Ausgangspunkt jeder Aktion im Programm. Er stellt somit den Haupt-Task dar.

Nur er hat den Zugriff auf die SD-Karte, den „ENROT“-Knopf, die GPIO-Taster und das LC-Display. Der Menu-Task hat die Aufgabe mit dem Benutzer zu interagieren und die Daten einer ausgewählten WAVE-Datei, befindlich auf der SD-Karte, in die Sound-Data-Queue zu kopieren. Wenn das Programm das erste Mal startet, muss das Verzeichnis der SD-Karte eingelesen werden. Dies geschieht durch Drücken des Tasters links vom Drehknopf. Wenn man jetzt den Drehknopf dreht, wird der Inhalt des Verzeichnisses im Display dargestellt, wobei der Titel in der oberen Zeile abgespielt werden kann. Dazu muss man nur den Drehknopf drücken. Während des Abspielens können die Tracks weiter gescrollt werden. Hier wird der Puffer der Queue ausgenutzt. Die Queue hat insgesamt 10 Einträge. Ist die Queue vollständig gefüllt geht der Menu-Task in den „blocked“-Zustand. Nur wenn ein Eintrag durch den Sound-Task entnommen wurde, wird der Menu-Task wieder aktiv und kann die Queue füllen. Genau das ist der Zeitraum, in dem das Menü aktualisiert werden kann. Scrollt man im Menü hoch und runter, wird das Display aktualisiert. Das beutet, dass das Befüllen der Queue leicht unterbrochen wird. Gäbe es keinen Puffer, würde die Soundausgabe gestört werden.

Um zu erfahren, welches Lied der Anwender durch die Menüsteuerung ausgewählt hat, wird einfach jede Aktion des Benutzers dokumentiert. Die Funktion `get_rotary_count()` zählt jeden Drehvorgang mit. Ist das erste Lied oder das letzte Lied angewählt, wird nicht mehr mitgezählt, auch wenn der Benutzer weiter dreht. Führt der Anwender einen Drehvorgang nicht ganz aus, sodass der Kontakt dauerhaft besteht, passiert so lange nichts, bis der Drehvorgang beendet wurde.

Drückt man erneut den Drehknopf, wird das aktuelle Lied gestoppt. Durch nochmaliges Drücken wird das Lied in der oberen Zeile des Display wieder ausgegeben. Durch Drücken des Tasters oberhalb des Drehknopfes, wird man aufgefordert, ein Lied über die serielle Schnittstelle zu senden. Ist der Sendevorgang erfolgreich, wird das Lied automatisch abgespielt.

Der Menu Task kümmert sich vollständig um das Einlesen der Daten von der SD-Karte und sendet diese in die Queue, wenn der Drehknopf gedrückt wird. Damit es keine zeitlichen Probleme bei der Ausgabe gibt, muss die SD-Karte mit mindestens 2 MHz betrieben werden.

5.2.2 Sound-Task

Der Sound Task hat die einzige Aufgabe PCM-Daten von der Queue an die Soundkarte weiterzugeben, und zwar immer dann, wenn ein Interrupt vom Soundmodul kommt. Der Sound-Task prüft kontinuierlich, ob ein Interrupt ausgelöst wurde. In der `Sound_ISR()` wird eine globale Interrupt-Variable gesetzt. Diese wird vom Sound-Task nach dem Einlesen wieder zurückgesetzt. Wurde ein Interrupt ausgelöst, holt sich der Sound-Task einen Eintrag aus der vollen Queue. Dieser Eintrag besteht aus 512 Samples, welche an den momentan von der Hardware nicht verwendeten Speicherbereich geschrieben werden. Außerdem werden auch 2 Adresszeiger übergeben, die auf die allozierten Speicherbereiche zeigen. Listing 14 zeigt die Form eines Eintrags der Queue.

```
struct sound_config
{
 unsigned char *adr_puffer1;
 unsigned char *adr_puffer2;
 unsigned char buffer[PUFFER_SIZE];
};
```

Listing 14: Struktur, die der Sound-Task benötigt

5.2.3 Error-Task

Der Error-Task bekommt vom Menu-Task in regelmäßigen Abständen durch die Variable `watchdog` ein durch FreeRTOS definiertes `pdPASS`. So weiß er, dass alles funktioniert. Wenn das so ist, lässt er eine LED alle 3 Sekunden blinken. Ist jedoch ein Task abgestürzt oder hat einen Bereich erreicht, den er nicht erreichen darf, so blinkt die LED alle 0,5 Sekunden. Sollte das gesamte System abstürzen, blinkt gar nichts mehr.

Listing 15 zeigt die Funktion, die für die Überprüfung zuständig ist. Ein Fehler wird angezeigt, wenn der Menu-Task abgestürzt ist oder wenn ein Task einen Bereich erreicht hat, den er nie erreichen dürfte. Wenn alles in Ordnung ist, wird der Watchdog zurückgesetzt.

```
static portBASE_TYPE prvCheckOtherTasksAreStillRunning( void )
{
      static portBASE_TYPE xAllTestsPass = pdTRUE;

      if( menu__error_status != pdPASS ) xAllTestsPass = pdFALSE;  //Fehler
      if( watchdog != pdPASS )            xAllTestsPass = pdFALSE; //Fehler

      watchdog = pdFALSE; // watchdog zurücksetzen
      return xAllTestsPass;
}
```

Listing 15: Watchdog-Implementierung

5.3 Verzeichnisstruktur, benötigte Dateien

Neben den FreeRTOS-Dateien sind für ein lauffähiges „WATOS“-Projekt die Dateien aus Tabelle 14 nötig.

Treiberdateien:	
easy_spi.c	Funktionen zum Ansprechen des SPI Modules
easy_spi.h	Header Definitionen
lcd.c	Funktionen zum Steuern des LC-Displays
lcd.h	Header Definitionen
rotary.c	Funktionen zum Auslesen des Rotary Registers
rotary.h	Header Definitionen
time.c	enthält delay Funktionen, initialisiert und steuert Timer 1
time.h	Header Definitionen
Programmdateien:	
fat.c	High Level FAT16 Funktionen
fat.h	Header Definitionen
mmc.c	Low Level SD Funktionen
mmc.h	Header Definitionen
soundmaster_uart.c	Funktionen zum Einlesen von Daten via RS232
soundmaster_uart.h	Header Definitionen
main.c	Hauptprogramm (Bild 24)

Tabelle 14: Verzeichnisstruktur „WATOS“-Player

5.4 „FAT16“- Implementierung

5.4.1 Überblick

Für die SD-Karte wurde das geläufige „FAT16“-Dateisystem von Microsoft ausgewählt. Es ist relativ einfach aufgebaut und im Internet existieren viele fertige, anpassbare Lösungen.

„FAT16“ wurde 1983 von Microsoft entwickelt. Es handelt sich um ein einfaches Dateisystem, das eine maximale Datenträgergröße von 4 GB zulässt. Spricht man von der FAT (File Allocation Table), so ist die Dateizuordnungstabelle gemeint. In ihr findet man die Startadressen der auf dem Datenträger befindlichen Dateien [21].

Für dieses Projekt wurde die „FAT16“-Implementierung für einen AVR Controller von Ulrich Radig ausgewählt [20]. Das Ansprechen des Dateisystems hat Ulrich Radig in seiner Implementierung mit High- und Low-Level Funktionen realisiert. Als Low-Level bezeichnet man die Funktionen, die

direkt mit den SD-Karten-Treibern kommunizieren. Sie steuern die eigentliche Übertragung der Bytes. Die High-Level Funktionen implementieren das Dateisystem und verwenden dazu die Low-Level Funktionen.

Die Low-Level Funktionen in der Datei `mmc.c` mussten komplett überarbeitet werden, um sie an das „Easy and Fast"-SPI-Modul anzupassen. Die High-Level-Funktionen aus der `fat.c` wurden größtenteils bis auf einige plattformabhängige Änderungen übernommen.

Beide Dateien wurden threadsicher gemacht. Es kann vorkommen, das der Menu-Task es nicht ganz schafft in seiner Zeitscheibe die Queue zu füllen, z.B. weil der Benutzer scrollt. Dadurch wird der Task an irgendeiner Stelle unterbrochen. Es muss dafür gesorgt werden, dass das nicht an einer kritischen Code-Stelle passiert.

5.4.2 „FAT16"-Aufbau:

Wichtige Kenngrößen sind Cluster und Sektoren. Jeder Sektor hat genau 512 Byte. Die SD-Karte wurde so formatiert, dass sie pro Cluster 64 Sektoren hat. Da jede Datei mit dem Anfang eines Clusters verbunden ist und nicht verbrauchte Sektoren mit 0x00 aufgefüllt werden, braucht eine 1 Byte große Datei mindestens 32768 Byte auf dem Datenträger. Vor dem Auslesen einer Datei muss also erst deren Anfangscluster und Größe ermittelt werden. Die Größe muss dann auf die Anzahl der Sektoren umgerechnet werden. Unter der Angabe des Startcluster und der Anzahl der belegten Sektoren kann eine Datei Stück für Stück ausgelesen werden.

Da die Adressierung mit Sektoren gestaltet ist, muss auf dieses Maß immer zurückgerechnet werden. Es kann immer nur ein Sektor nach dem anderen ausgelesen werden. In der Init-Funktion wird Sektor 0 ausgelesen. Hier befindet sich der Master Boot Record (MBR). Er enthält Informationen über die Partitionen, die auf dem Datenträger sind. Aus dem MBR wird der Startsektor der Partition ausgelesen. An diesem Sektor befindet sich dann der Volume Boot Record (VBR) oder auch Bootsector. Dieser enthält viele wichtige Größen, um mit der Partition arbeiten zu können, z.B. Bytes pro Sektor, Sektoren pro Cluster, Anzahl reservierter Sektoren, Anzahl der FAT Kopien, Sektoren pro FAT usw..

Abbildung 25 zeigt die absoluten Adressen aus dem VBR, bezogen auf die verwendete SD-Karte. Da nur eine Partition verwendet wird, hat der MBR auch nur einen Eintrag.

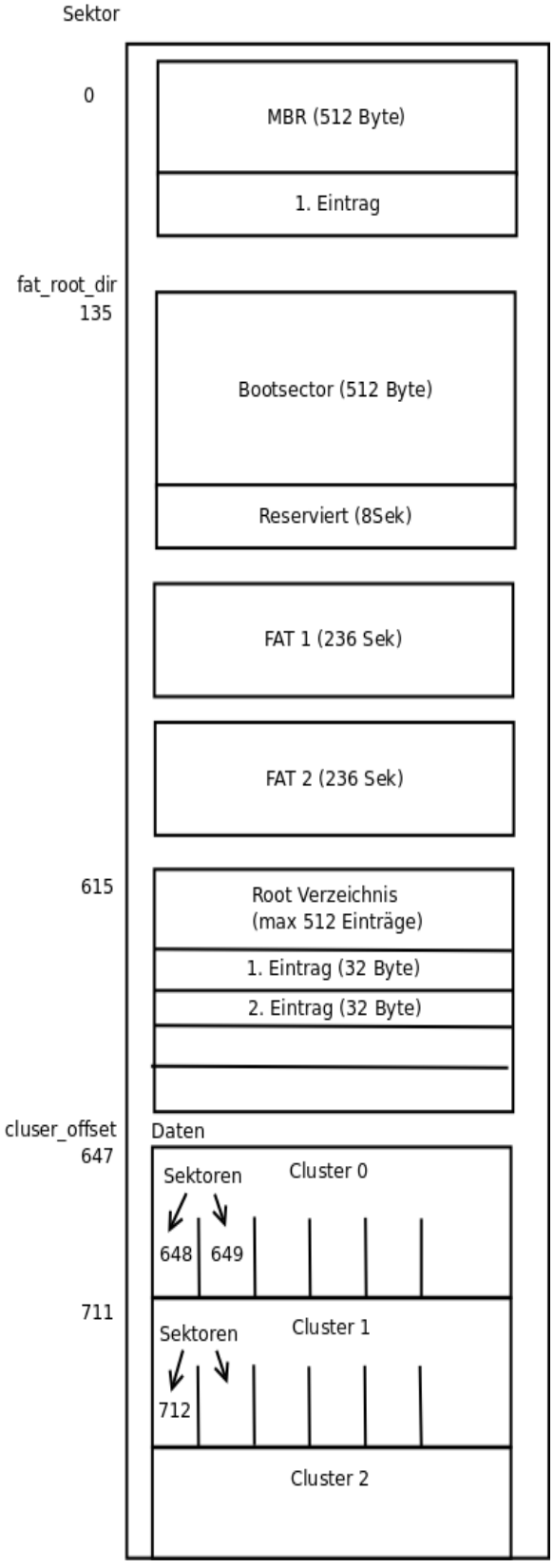
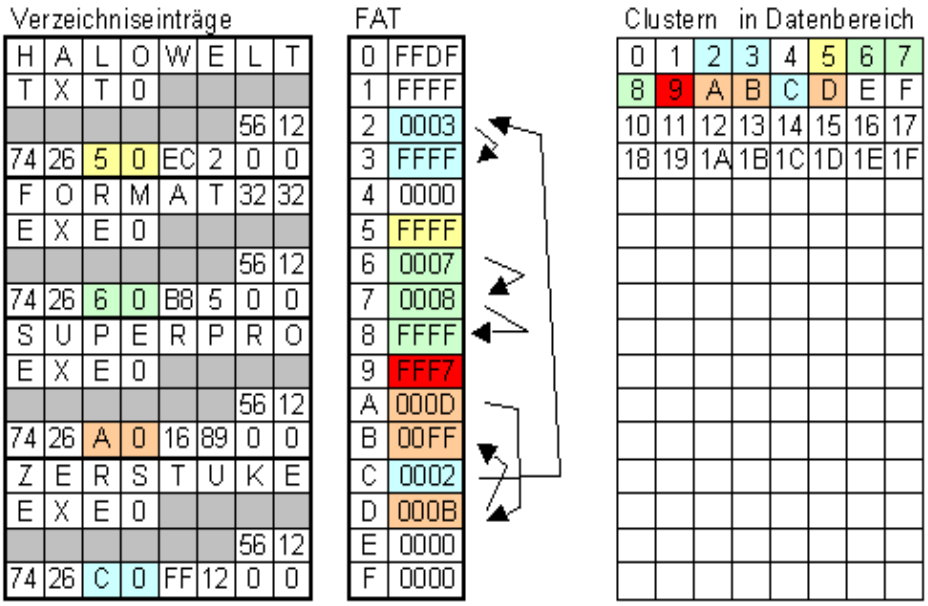

Sektor
0
MBR (512 Byte)
1. Eintrag
fat_root_dir
135
Bootsector (512 Byte)
Reserviert (8Sek)
FAT 1 (236 Sek)
FAT 2 (236 Sek)
615
Root Verzeichnis
(max 512 Einträge)
1. Eintrag (32 Byte)
2. Eintrag (32 Byte)
cluser_offset
647
Daten
Sektoren
Cluster 0
648
649
711
Sektoren
Cluster 1
712
Cluster 2

Abbildung 26 zeigt einen vergrößerten Teilabschnitt der Abbildung 25. Die Verzeichniseinträge (Root-Verzeichnis) sind ganz links, die FAT (Zuordnungstabelle) in der Mitte und die Cluster im Datenbereich ganz rechts.

Will man eine Datei auslesen, so muss erst im Root-Verzeichnis nachgesehen werden, in welchem Cluster die Datei startet. Nehmen wir die Datei `format.exe`. Die Datei startet bei Cluster 6. In der FAT unter dem Index 6 steht dann der nächste Cluster, hier Cluster 7. Unter dem Index 7 in der FAT findet man den nächsten Cluster 8. Unter Index 8 schließlich endet die Datei, da der Cluster mit 0xFFFF abgeschossen ist. Die Datei besteht also aus insgesamt 3 Clustern, nämlich Cluster 6, 7 und 8. Diese können jetzt aus dem Datenbereich ausgelesen werden.

Unterverzeichnisse werden nicht unterstützt.



5.4.3 Funktionsweise der High- und Low-Level Funktionen:

5.4.3.1 Die High-Level-Implementierung

In der Datei `fat.c` befinden sich die High-Level Funktionen für das Hauptprogramm. Die wichtigsten Funktionen sind `fat_read_file()`, `fat_write_file()` und `fat_init()`. Sie sind die Schnittstellen zum Hauptprogramm.

Ablauf um eine Datei einzulesen:

Mit der Funktion `fat_read_file(int Cluster, char *Buffer, int sectornummer)` wird der Anfangscluster der Datei, ein Ziel-Buffer und die Sektornummer, die ab dem Anfangscluster ausgegeben werden soll übergeben. Die Funktion wird so oft aufgerufen, wie Sektoren ausgelesen werden sollen.

Um die Datei einzulesen, benötigt man alle Sektoradressen dieser Datei. Die Low-Level-

Implementierung kann nur mit Sektoradressen arbeiten. Um die Sektoradressen zu erhalten, muss ein wenig gerechnet werden. Der folgende Ablauf zeigt die nötigen Schritte.

In der folgenden Berechnung wird der Cluster ausgerechnet, indem sich der gewünschte Sektor befindet. Mögliches Ergebnis wäre z.B. der dritte Cluster einer Datei:

```
Block = (BlockCount/cluster_size);  // welcher cluster
```

Die Funktion `fat_load()` prüft, in welchem Cluster (absolut) sich der momentan auszulesende Sektor befindet. Dabei muss die FAT durchlaufen werden. In der Variablen **`Block`** steht dann der Cluster, der momentan relevant ist:

```
fat_load (Cluster,&Block,Buffer)
```

Beispiel: Eine Datei ist 204800 Byte groß, das heißt sie belegt 400 Sektoren. Man weiß den Startcluster der Datei, z.B. Nr. 10, und dass ein Cluster 64 Sektoren enthält. Will man also z.B Sektor 65 auslesen, so muss man erst in der FAT nachsehen wo der nächste Cluster ist. Dies könnte z.B. die Nummer 22 sein. In diesem Cluster ist dann der erste Sektor auszulesen.

Konkret ist das Ganze komplizierter, weil ständig zwischen Byte-, Cluster- und Sektoradressen umgerechnet werden muss. Will man z.B. den Cluster Nr. 7 aus der FAT auslesen, muss erst der absolute Sektor ausgerechnet werden und dann die Byte-Adresse (siehe Sektornummerierung aus Abbildung 25). Im Abschnitt 9.1.9 befindet sich ein detailliertes Beispiel wie genau die FAT ausgelesen wird.

Um die FAT auszulesen berechnet die `fat_load()`-Funktion zuerst in welchem Sektor der FAT sich der Eintrag befindet:

```
FAT_Block_Addresse = ((Cluster*2) / BlockSize) + volume_boot_record_addr +
fat_offset
```

*Beispiel: Eine Datei beginnt im Cluster 513. Jetzt möchte man wissen wo sich der nächste Cluster befindet. Mit `((Cluster*2) / BlockSize)` rechnet man also den nächsten Eintrag in der FAT aus. Das Ergebnis ist (513*2)/512 = 2. Der zweite Sektor der FAT gibt Aufschluss über den nächsten Cluster der Datei.*

Der Sektor wird nun eingelesen. Da die FAT pro Eintrag 2 Byte benötigt und jede Adressierung auf Byte-Adressen umgewandelt werden muss, ist ein weiterer Schritt nötig. Der oben genannte Cluster 513 wird nun auf seine Byte-Adresse im zweiten Sektor der FAT umgerechnet. Ergebnis: (513*2)%512 = 2. Das bedeutet; Das 2te Byte im 2ten Sektor der FAT ist das LOW-Byte, das dritte das HIGH-Byte des nächsten Clusters:

```
FAT_Byte_Addresse = (Cluster*2) % BlockSize;
```

Da die FAT eine verkettete Liste ist (Clusterkette) muss diese so lange durchlaufen werden,bis man den Cluster, in dem der relevante Sektor steht erreicht hat. Hat man schließlich den richtigen Cluster in der FAT gefunden, von dem ein Sektor gelesen werden soll, muss die Clusternummer auf eine absolute Sektoradresse umgerechnet werden. Das wird mit der anschließenden Berechnung realisiert:

```
Block = ((Block-2) * cluster_size) + cluster_offset;
// Cluster im Datenbereich = Clusternummer FAT - 2
```

Mit der letzten Berechnung muss man noch konkret die Nummer des Sektors, die man in dem Cluster wissen will auf die absolute Sektoradresse des Clusters aufaddieren. Mit der Modulo Rechnung bekommt man immer den Rest ab Anfang des Clusters, der aufaddiert werden muss:

```
Block += (BlockCount % cluster_size); // aufaddieren des blockcount
```

Jetzt kann der Sektor mithilfe der LOW-Level Funktion `mmc_read_sector()` in den Buffer eingelesen werden:

```
mmc_read_sector (Block,Buffer);
```

Bevor man eine Datei einlesen kann, muss man den Startcluster und die Größe der Datei ermitteln. Dafür stehen die Funktionen `fat_read_dir_ent()` und `fat_search_file()` zur Verfügung. In beiden Fällen wird das Root-Directory durchkämmt. Die erste Funktion liest das komplette Root-Directory mit allen Dateiattributen aus.

Ablauf um das Verzeichnis auszugeben:

Um mit `fat_root_dir_adr()` ein Verzeichnis auszulesen benötigt man zuerst die Sektornummer des Root-Verzeichnises. Dafür muss der VBR der aktuellen Partition ausgelesen werden. Die Adresse des ersten VBR wird nach der Ausführung der `fat_init()`−Funktion aus dem MBR heraus gelesen, der immer bei Sektor 0 beginnt. Hat man die VBR Adresse, so addiert man jeweils die Größen der FAT Tabellen, reservierten Sektoren und die des Bootsektors auf. So erhält man den Anfangssektor des Root-Verzeichnis. Die Größen erhält man am Anfang des Initialisierungsprozesses der alle relevanten Daten aus dem MBR / VBR herausliest:

```
fat_root_dir_addr (unsigned char *Buffer)
```

Jetzt ist es mit der folgenden Funktion möglich, das komplette Verzeichnis auszulesen. Übergibt

man `dir_cluster = 0`, wird das gesamte Verzeichnis gelistet. Mit dem Argument `entry_count` gibt man die Position aus die man gerne hätte. „FAT16“ unterstützt max 512 Root-Verzeichnisse, weshalb die Größe des Root-Verzeichnises auf 16kB fixiert ist:

```
fat_read_dir_ent (cluster, entry_count, *size, *dir_attrib, *buffer)
```

5.4.3.2 Die Low-Level-Implementierung

In der Datei `mmc.c` befinden sich die LOW-Level-Routinen zum Ansprechen der SD-Karte. Die wichtigsten Funktionen sind: `mmc_read_sector()` , `mmc_write_sector()` und `mmc_init()`. Sie sind die Schnittstellen zu den High-Level Routinen aus der Datei `fat.c`. Die Aufgabe der Low-Level Routine ist es, via SPI mit der SD-Karte zu kommunizieren. Die Sektoradressen werden dazu in Byte-Adressen umgewandelt, und der 512-Byte große Inhalt der Adresse wird zurückgegeben. Die Übertragung der einzelnen Bytes wird mit den Treibern der „Easy and Fast“-SPI-Core ausgeführt.

Ablauf um einen Sektor einzulesen:

Mit der Funktion `mmc_read_sector(int addr, char *Buffer)` wird der angegebene Sektor in den Buffer geschrieben. Da die SD-Karte selber in Bytes adressiert ist, muss vom angegebenen Sektor auf das richtige Byte umgerechnet werden:

```
addr = addr << 9; //addr = addr * 512
```

Die errechnete Adresse muss dann in den CMD Lesebefehl eingefügt werden:

```
cmd[1] = ((addr & 0xFF000000) >>24 );
cmd[2] = ((addr & 0x00FF0000) >>16 );
cmd[3] = ((addr & 0x0000FF00) >>8 );
```

Dann kann der Befehl an die Karte gesendet werden (Read-Single-Block: CMD17 {0x51,0x00,0x00,0x00,0x00,0xFF}) siehe Abschnitt 4.2.2 .

In diesem Fall wird ab der Adresse 0x00 eingelesen. CMD[3] (LSB) wird nicht beschrieben, da die kleinste Einheit, die ausgelesen wird, 512 Byte (ein Block) sind. Durch diese Art der Adressierung können maximal 4 GB angesprochen werden.

Die Karte lädt den Block in ihren internen Zwischenspeicher, wo er mit der eingestellten Frequenz ausgelesen wird; so lange wird 0x00 zurückgegeben. Wenn der Vorgang fertig ist, gibt die Karte das Data-Token 0xfe und den angefragten Block, gefolgt von zwei CRC Bytes, zurück.

5.5 High-und Low-Level API

Tabelle 15 gibt einen Überblick über alle High- und Low-Level Funktionen, die es gibt. Eine genaue Beschreibung der jeweiligen Funktion ist im Quelltext zu finden.

Datei	Funktion	Beschreibung
mmc.c	mmc_write_command()	Sendet Kommando an die SD-Karte (z.B. Lesen von Adresse)
	mmc_init()	Initialisiert die SD-Karte
	mmc_write_sector()	Erzeugt das Kommando zum Adressieren eines Sektors für den Schreibvorgang von 512 Byte.
	mmc_read_block()	Ein 512 Byte großer Sektor wird gelesen
	mmc_read_sector()	Erzeugt ein Kommando zum Adressieren eines Blocks der somit gelesen werden kann. Ruft mmc_read_block() auf.
	mmc_read_cid()	Liest das CID-Register
	mmc_read_csd()	Liest das CSD-Register
fat.c	fat_init()	Initialisiert das FAT16, liest alle nötigen Daten aus MBR/VBR
	fat_root_dir_addr()	Sektor-Adresse des Root-Verzeichnisses ermitteln
	fat_dir_entr()	Gibt das Root-Verzeichnis aus
	fat_load()	Sucht in der FAT zu einem Sektor dessen Clusternummer, dabei wird die Clusterkette der FAT durchkämmt
	fat_read_file()	Liest einen bestimmten Sektor einer Datei ein
	fat_write_file()	Schreibt einen bestimmten Sektor einer Datei
	fat_search_file()	Sucht den Startcluster, Größe, Attribute einer bestimmten Datei

Tabelle 15: High- und Low-Level-API des Dateisystems

5.6 Software Low-Level-SPI-Lösung (alternativ)

Es gibt auch die Möglichkeit das SPI-Protokoll via Software zu implementieren. In der Datei `mmc.c` findet man die Funktionen dazu als Kommentar. Diese Lösung ist aber viel zu langsam, da nur eine Frequenz von ca. 400 kHz erreicht wird. Listing 16 zeigt die Code-Implementierung.

```
unsigned char mmc_read_byte (void)
{
unsigned char a, Byte = 0;
for (a=8; a>0; a--) //das Byte wird Bitweise nacheinander empfangen MSB First
 {
  *MMC_Write &=~(1<<SPI_Clock); //erzeugt ein Clock Impuls (Low)
  if (bit_is_set2(SPI_DI) > 0){ //Lesen des Pegels von MMC_DI
    Byte |= (1<<(a-1));
    }
  else { Byte &=~(1<<(a-1)); }
  *MMC_Write |=(1<<SPI_Clock); //setzt Clock Impuls wieder auf (High)
 }
return (Byte);
}
```

Listing 16: Software-SPI Lösung

Ablauf:

Es wird nach jedem erzeugtem Takt-Impuls ein Bit von der MISO Leitung eingelesen. Das Ganze läuft 8 mal ab, dann wird das eingelesene Byte zurückgegeben.

6. „WATOS“-Player ohne PC betreiben

Es wäre es aus Demonstrationsgründen von Vorteil, alle Hardware- und Softwaredaten des fertigen Projektes auf dem Board zu speichern, um nicht jedesmal vorher alles mit dem PC übertragen zu müssen. Die Software-Daten lassen sich in einem der Flash Bausteine auf dem Board speichern. Der Verfasser hat sich für den unter 2.1 beschriebenen 32 Mbit parallel Flash (4 MByte) entschieden, da er den größten Speicherplatz bietet.

Damit die Daten vom Flash Baustein in den DDR2 RAM übertragen werden können, benötigt man den EDK Bootloader. Der Bootloader befindet sich im Block-RAM des SoC und wird daher mit der Hardwaredatei abgespeichert.

Die Hardwaredatei (Bitstream) wird in dem kleinsten Flash Baustein 8 Mbit parallel Flash (500 KByte) gespeichert. Die Einrichtung eines solchen Systems ist im Anhang unter Abschnitt 9.1.5 zu finden.

Ablauf:

Nach dem Einschalten des Systems werden die Hardwaredaten (SoC + Bootloader im Block RAM) vom parallel Flash zum Konfigurieren des FPGAs verwendet. Danach wird die neue Hardware im FPGA zur Ausführung gebracht. Diese startet den Bootloader, der das Programm vom Flash in den DDR2 kopiert und schließlich zur eigentlichen Programm-Startadresse springt.

7. Praktikumsaufgaben

Da die hier vorliegende Arbeit -wie in der Motivation aufgezeigt- auch einen Grundstein für die Einarbeitung in SoC-Systeme bieten soll, sind hier beispielhafte Aufgabenstellungen aufgeführt. Die Aufgabenstellungen bauen aufeinander auf, sie sollten daher der Reihe nach bearbeitet werden. Der zeitliche Rahmen der Bachelorarbeit hat es aber nicht mehr zugelassen die Aufgaben auf die Durchführbarkeit in einem angemessenen Rahmen zu testen.

7.1 Mögliche Aufgabenstellungen

1. Serielle Schnittstelle

Über die serielle Schnittstelle ist es möglich WAVE-Dateien einzulesen und in den DDR-Speicher zu schreiben. Die Übertragung endet, sobald 0x03 erkannt wurde. Bei der Übertragung ist also darauf zu achten, dass **nur** am Ende der PCM-Daten die Terminal-Kennung 0x03 vorkommt

Bearbeiten Sie die Funktion `serial_load()` in der Datei `soundmaster_uart.c` so, dass zuerst der Header der WAVE-Datei eingelesen wird. Daraus soll die Größe der Datei ausgelesen werden. Die Übertragung endet, wenn die Daten komplett übertragen wurden. Auf die Terminal-Kennung 0x03 kann jetzt verzichtet werden.

2. SPI Modul

Ändern Sie die Hardware des SPI-Moduls so ab, dass es möglich ist, die Frequenz höher als 16,2 MHz einzustellen. Ferner soll ein 16 Bit Übertragungsmodus eingebaut werden. Das SPI-Modul soll mehrere SPI-Slave-Geräte adressieren können.

Ändern Sie auch die Treiber in der Datei `easy_spi.c`. Implementieren Sie 2 neue Funktionen `mmc_read_word(), mmc_write_word()`. Die Funktion `mmc_set_divider()` soll in `mmc_init()` abgeändert werden. Hier werden nun der Teiler, die Anzahl der Slaves und die Datenbreite eingestellt. Die Anzahl der Slaves und die Datenbreite speichern sie in der Hardware im Control-Register. Die Funktion `mmc_enable()` und `mmc_disable()` soll als Argument die Slave-Nummer enthalten, die aktiviert/deaktiviert werden soll.

3. SPI-Modul, SPI-Flash

Die an der seriellen Schnittstelle übertragenen Daten sollen jetzt in einem der beiden SPI-Flash Module gespeichert werden. Verwenden Sie dafür das "Easy and Fast"-SPI-Modul. Beim Drücken

des N-Buttons soll die Datei in den Flash übertragen werden. Beim Drücken des O-Button soll die Datei aus dem Flash wiedergegeben werden.

Dafür ändern Sie die Funktion `serial_load()` abermals. Die Funktionen zum Ansprechen des SPI Modules sind unter Abschnitt 4.2 erklärt.

Das Hauptprogramm muss um eine Button-Abfrage erweitert werden.

Auf der Seite 93 im Benutzerhandbuch finden Sie Informationen zu den SPI-Flash-Modulen.

Ändern Sie die Datei `system.ucf` so ab, dass SD-Karte sowie Flash die gleichen Leitungen haben.

Bsp.:

```
## SD CARD on Port J20
NET "fpga_0_MOSI_pin" LOC = "W16"|LOC = "AB14"| IOSTANDARD = LVCMOS33 ;
...
```

Außerdem benötigen sie zwei weitere Leitungen:

```
dataflash_wp  (C14)     (write protect)
dataflash_rst (C15)     (reset)
```

Diese sollen auch in das Control-Register aufgenommen werden, um Sie von dort aktivieren zu können.

Verwenden sie den 16Mbit **AT45DB161D** Flash-Baustein. Setzen sie dazu den Jumper (J1) richtig. Laden sie sich das Datenblatt des Bausteines herunter und studieren sie seine Steuerbefehle.

Implementieren sie Funktionen zum Initialisieren, Schreiben und Lesen des SPI-Flash-Bausteines. Nehmen Sie dazu die Funktionen aus der `mmc.c` als Vorlage. Erstellen Sie dafür 2 neue Dateien: `spi_flash.c`, `spi_flash.h`.

4. Display-Anzeige

Während einer Soundausgabe soll das Display rückwärts die Laufzeit des Titels herunterzählen. Dazu implementieren Sie einen neuen Task, der über eine Queue versorgt wird. Die Struktur, die der Queue übergeben wird, finden sie in der `lcd.h`.

```
struct display_text
{
portBASE_TYPE dauer;
char *titel;
char ausgabe;
};
```

Listing 17: Struktur des Queue-Eintrags

Sobald ein Titel ausgewählt wurde, wird eine Nachricht mit Titel und Dauer der Queue übergeben. Der Display-Task soll erst aufwachen, wenn er eine Nachricht bekommen hat. Dazu stellen sie die Priorität des Display-Tasks auf 2. Die Queue soll fünf Strukturen aufnehmen können.

```
static void display_task( void *pvParameters )
{
    for(;;)
    {
      if(xQueueReceive(display_Queue, &display_buffer, portMAX_DELAY))
      {
            // Code
      }
}
```

Listing 18: Grundsätzlicher Aufbau des Display-Task

Der Display-Task soll jetzt mithilfe der API-Funktion `vTaskDelayUntil()` das Display jede Sekunde dekrementieren. Alle Ausgaben des Menu-Tasks, sollen auch über den Display-Task ausgegeben werden, um konkurrierenden Zugriff auf das LCD zu vermeiden. Will man nur eine normale Ausgabe, so setzt man die Variable `ausgabe` in der Struktur `display_text` auf Eins. In diesem Fall befindet sich in der Variable `titel` der Ausgabestring.

8. Zusammenfassung / Fazit

Im Rahmen der Bachelorarbeit wurde untersucht, ob sich das FreeRTOS-System auf das Spartan-3a Development Board installieren lässt und ob es sich für die Ausarbeitung von Praktikumsaufgaben eignet. Dabei ist ein Grundstock für eine weitere Entwicklung gesetzt worden. Außerdem wurden im „WATOS“-Anwendungsprogramm die Echtzeitfähigkeit, Synchronisationsverfahren und das Interagieren mit kritischen Abschnitten überprüft.

Um diese Aufgabenstellung zu realisieren, wurden drei eigene Hardwaremodule entwickelt. Die Hardwaremodule wurden alle als sogenannte „Soft-Cores“ mit den Xilinx-EDK Werkzeugen erstellt. Als Programmiersprache wurde VHDL verwendet.

Die Hardwaremodule bestehen aus einem SPI-Core, einem LCD-Core und einem Core für den Drehbutton. Das bereits vorhandene Audio-Modul, „PCM-Master-8“ wurde übernommen und im Zuge dieser Arbeit um eine Interruptfähigkeit erweitert. Entwickelt wurde es im Praktikum „Systems-on-a-Chip“ unter Herrn Prof. Dr. Kiefer von Florian Richter und dem Autor dieser Arbeit. Das SPI-Modul wurde entworfen, um eine SD-Karte einfach und schnell anzusteuern. Damit es unter FreeRTOS angewendet werden kann, wurden dafür „thread-sichere“ Treiber entwickelt. Das gleiche gilt für den Drehknopf, der hardwaremäßig entprellt wurde, und für das LC-Display. Beim LC-Display wurde die komplette Ansteuerung via Software gestaltet. Nur für die Abbildung der Hardware-Slave-Register auf die Hardwarepins wurde ein Hardwaremodul erzeugt. Die Treiber für das Soundmodul wurden ebenfalls komplett selbst entwickelt.

Das Anwenderprogramm („WATOS“- Player) wurde mit Hilfe von FreeRTOS realisiert. Dabei verteilt sich Die Anwendung auf insgesamt 3 Tasks. „WATOS“ hat die Aufgabe, eine Benutzerinteraktion zu schaffen und den Soundcore mit Musikdaten zu versorgen. Um konkurrierende Zugriffe auf Hardware-Ressourcen zu vermeiden, wurde eine Queue eingesetzt. „WATOS“ reagiert auf Interrupts des Soundcores. Dafür wurde ein Interrupt-Handler registriert. Da der MicroBlaze nur einen Interrupt-Eingang hat, es aber zwei Quellen (Soundcore und Timer) gibt, musste ein Interrupt-Controller eingebaut werden.

Damit man komfortabel Daten (WAVE-Dateien) auf die SD-Karte laden kann, unterstützt der „WATOS“-Player das Dateisystem „FAT16“. Diese Dateien werden am LC-Display angezeigt. Durch scrollen wählt man den gewünschten Titel aus und spielt ihn mit einem Druck auf den Drehknopf ab.

Durch diese Arbeit ist nicht nur ein lauffähiger Demonstrator für ein SoC auf einem FPGA-Board entstanden, sondern auch ein gut dokumentierter Einsteig in die Entwicklung von Hardware-Treibern und eigener Hardwaremodule. Außerdem wurden essentielle Eigenschaften von Betriebssystemen erklärt und verwendet. Der genau dargestellte Initialisierungsprozess einer SD-Karte im Anhang gibt einen guten Einblick in die Funktionsweise des SPI-Protokolls. Nicht zuletzt bietet die Arbeit durch das ausführliche Kapitel 5.3 einen Einstieg in das Verständnis des Aufbaus von Dateisystemen.

So können Studenten durch das Bearbeiten der Praktikumsaufgaben sofort in die Welt der SoC einsteigen.

9. Literaturverzeichnis

[1] XILINX *XTP013 EDK 10.1 Concepts Tools and Techniques* 18.09.2008
URL: www.xilinx.com

[2] XILINX *UG330 Spartan-3A FPGA Starter Kit User Guide* 21.06.2007
URL: www.xilinx.com

[3] Florian Richter *SoC Ausarbeitung* FH-Augsburg 2009

[4] XILINX *UG081(v9.0) Microblaze Processor Reference Guide* 17.01.08
URL: www.xilinx.com

[5] Mikrocontroller.net *AVR-GCC Tutorial* 20.09.09 URL: www.mikrocontroller.net

[6] Thorsten Horn *ASCII-,ANSI- u. HTML Zeichencodes 1998*
URL: http://www.torsten-horn.de/techdocs/ascii.htm

[7] XILINX *XAPP778 Using and Creating Interrupt-Based Systems* 01.11.2005
URL: www.xilinx.com

[8] XILINX *DS572 XPS Interrupt Controller Data Sheet* 02.12.2009
URL: www.xilinx.com

[9] IBM *White Paper CoreConnect Bus* 01.09.99
https://www01.ibm.com/chips/techlib/techlib.nsf/techdocs/
852569B20050FF77852569910050C0FB

[10] FREE RTOS: *Implementierung* 2010
URL: http://www.freertos.org/implementation/index.html

[11] FREE RTOS: *API Übersicht.* 2010 URL: http://www.freertos.org/a00106.html

[12] FREE RTOS: *Dokumentation zu Semaphoren. 2010*
URL: http://www.freertos.org/a00113.html

[13] FREE RTOS: *Konfiguration des Betriebssystems. 2010*
URL: http://www.freertos.org/a00110.html

[14] IBM: *Spezifikation des Processor Local Bus*
URL: http://www.01.ibm.com/chips/techlib/techlib.nsf/techdocs/
3BBB27E5BCC165BA87256A2B0064FFB4.

[15] Richard Barry *Using the FreerTOS Real Time Kernel, A Practical Guide* 2009

[16] EDK-9-Patch *Durchzuführende Änderungen* 16.6.2008
http://ares.mailus.de/~erik/freertos_microblaze_ise_9.1/freertos_5.0_changes.txt

[17] EDK-9-Patch *Fertig geänderte Daten* 16.6.2008
URL: http://ares.mailus.de/~erik/freertos_microblaze_ise_9.1/files/

[18] Cosmiac *IP Modul selbst erstellen* Stand: Mai 2010
URL: http://www.cosmiac.org/pdfs/04_Adding_IP_Lab.pdf

[19] Florian Richter, David Lucinkiewicz *PCM-Master-8 Soundcore* 2009 FH-Augsburg
URL:http://fpgalinux.informatik.fh-augsburg.de/projects/fpgalinux/browser/PCM-Master-8

[20] Ulrich Radig *FAT16 Implementierung* 12.06.2004 URL: http://www.ulrichradig.de/

[21] Anton Zechner *Das FAT Datei Format* 2004
URL: http://members.inode.at/anton.zechner/az/FatFormat.htm

[22] Michael Dworkin *Einführung in das FAT16/32 System* Stand: Mai 2010
URL: http://cc5x.de/MMC/FAT.html

[23] MorbZ-Bot *Der DLX Mikroprozessor* 12. Juni 2009
URL: http://de.wikipedia.org/w/index.php?title=DLX-Mikroprozessor&oldid=61072871

[24] FREERTOS: Homepage von SafeRTOS, IEC 61508 zertifiziert 2010
http://www.freertos.org/safertos.html

[25] Barry, Richard *License and Warranty* 2010
URL: http://www.freertos.org/a00114.html

[26] Round Robin
Andrew S. Tanenbaum *Moderne Betriebssysteme* Pearson Studium April 2009

[27] Niclas Winquist *Mutex vs. Semaphore The Toilet Example,* Stand: 2005
URL: http://pheattarchive.emporia.edu/courses/2009/cs557f09/hand07/Mutex%20vs_%20Semaphore.htm

[28] Determinismus (Algorithmus)
John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium. 2002. ISBN 3-8273-7020-5

[29] SD / MMC Card Adapter Item#: SMD-0100 Stand: 30.04.2010
URL: http://www.nkcelectronics.com/

[30] Serial Peripheral Interface
Klaus Wüst *Mikroprozessortechnik* VIEWEG+TEUBNER 3. Auflage 2009 S.251
ISBN:978-3-8348-0461-7

10. Anhang

10.1 Anleitungen und Beschreibungen

10.1.1 SVN-Repository

Ein SVN Repository ist unter:
http://freertos.informatik.fh-augsburg.de/projects/freertos
eingerichtet.

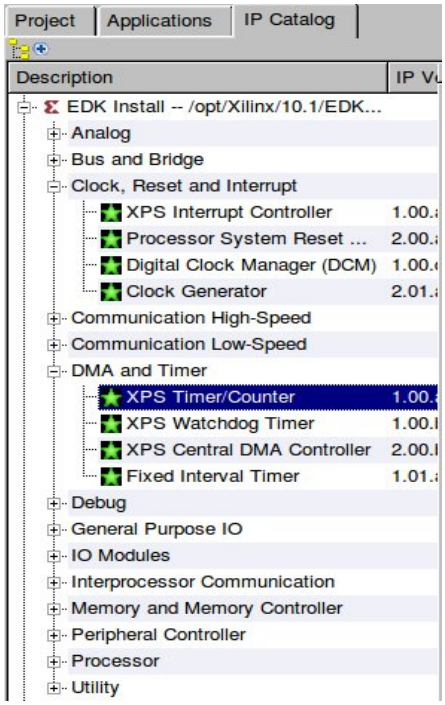
In diesem Repository befinden sich alle Projektdaten einschließlich Dokumentationen.
Unter folgendem Link lässt sich das Projekt auschecken:
http://freertos.informatik.fh-augsburg.de/svn/freertos/
Das hat Repository den gleichen Aufbau wie die Daten-CD:

Binaries	Hardware Image (Spartan 3a) / Software Image (FreeRTOS)
Dokumentation	Notizen, Bilder, Handbücher, Zeichnungen, dieses Dokument
FreeRTOS_ORG_Data	Original FreeRTOS Daten
Hardware	Selbst entwickelte Hardwaremodule
SOC PCM_Master _8	PCM Master 8 Hardwaremodul
Software	FreeRTOS APP (portiert), TestApps, Treiber für eigene IP Module
*.TAR	komplette EDK 10 Projekte

Tabelle 16: SVN-Repository-Aufbau

10.1.2 Einbinden der notwendigen IPs für FreeRTOS

FreeRTOS braucht einen Timer und, falls es mehrere Interrupt-Quellen gibt, einen Interrupt Controller. Unter IP-Catalog findet man im EDK alle erdenklichen IPs. Eingebunden wird der Timer und Interrupt Controller, über `add IP`.

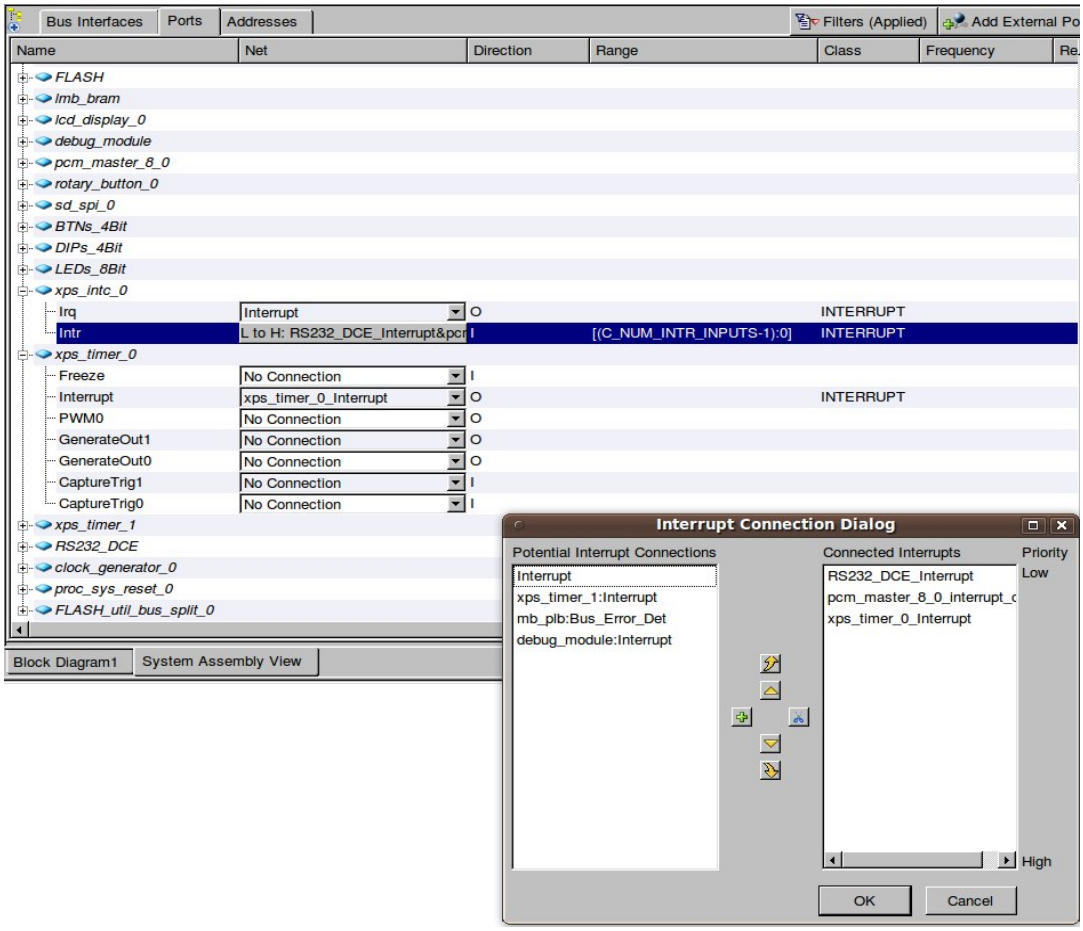


Als Nächstes muss alles richtig verdrahtet werden. Mit einem Rechtsklick auf `microblaze_0` erstellen wir einen neuen Interrupt-Eingangsport.

Das Gleiche wird beim Interrupt-Controller gemacht, nur wird hier der Ausgangsport mit dem Eingang des MicroBlaze verbunden.

Danach müssten alle Interrupt-Quellen am Interrupt-Controller angeschlossen werden. Dazu verbindet man den Interrupt-Ausgang jedes Gerätes mit dem Interrupt-Controller.

Im Interrupt-Controller müssen jetzt noch die Prioritäten der Interrupt-Quellen festgelegt werden, siehe dazu Bild 28.



Am Schluss müssen noch Adressen für die neuen Geräte vergeben werden. Dies erfolgt im Tab `Adresses` mit einem Klick auf `Generate Adresses`.

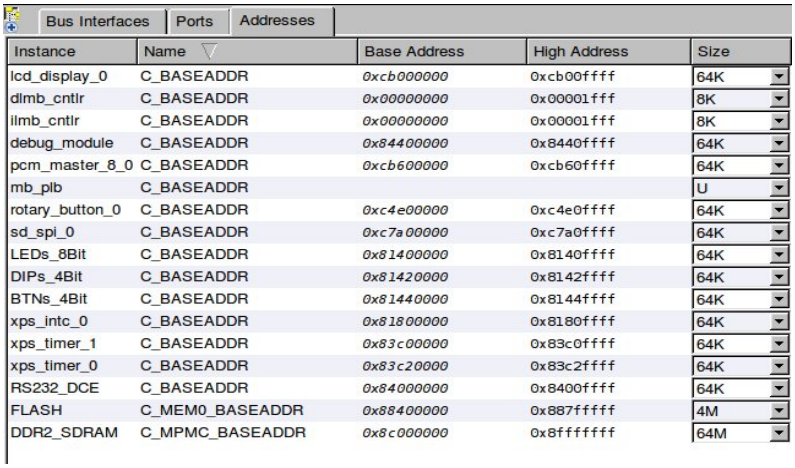


Abbildung 29: Adressen generieren

10.1.3 Erzeugung des FreeRTOS-Tick: port.c / portasm.s

Anhand des folgenden Quellcodes wird erklärt wie das System auf einen Interrupt, z.B. von einem Timer reagiert. Außerdem werden auch die beteiligten Codeausschnitte gezeigt. Dies ist unter anderem wichtig wenn man FREERTOS für eine komplett neue Platform portieren will.

portasm.s:

Der Compiler sucht selbständig nach dem "`_interrupt_handler`" String. Die Adresse wird dann in den Interrupt-Vektor nach 0x10 geschrieben, siehe 2.1.2 Interrupt. Dieser String bildet die Einsprungmarke für einen Interrupt, siehe Listing 19.

```
_interrupt_handler:
....
....
....
bralid r15, vTaskISRHandler
or r0, r0, r0
portRESTORE_CONTEXT
```

Listing 19: Einsprung für die Interrupt-Routine

port.c

Wird ein Interrupt ausgelöst, springt der `_interrupt_handler` in die Funktion `VTaskISRHandler()` die wiederum den Interrupt-Handler des Interrupt-Controllers (I.C.) aufruft, siehe Listing 20.

```
void vTaskISRHandler( void )
{
/*Call the Xilinx interrupt handler by the XPS Interrupt Controller */
XIntc_DeviceInterruptHandler(0);
//xil_printf("Handler\n");
}
```

Listing 20: Standard Interrupt-Routine ruft den Handler des Interrupt-Controllers auf

Durch die vorherige Registrierung des Timers am I.C. kann dieser entscheiden, ob und wohin gesprungen wird. Listing 21 zeigt eine solche Registrierung.

```
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
XPAR_XPS_INTC_0_XPS_TIMER_0_INTERRUPT_INTR, (XInterruptHandler) vTickISR,
(void*) XPAR_XPS_TIMER_0_BASEADDR);
```

Listing 21: Registrieren des Timers am Interrupt-Handler

Im Falle des Timers wird zur Funktion `VtickISR()` gesprungen. Schließlich wird in dieser Funktion der Tick-Count erhöht und ein Kontextwechsel veranlasst, siehe Listing 22.

```
void vTickISR( void *pvBaseAddress )
{
unsigned portLONG ulCSR;

/* Increment the RTOS tick - this might cause a task to unblock. */
vTaskIncrementTick();

/* Clear the timer interrupt */
ulCSR = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
XTmrCtr_mSetControlStatusReg( XPAR_XPS_TIMER_0_BASEADDR, portCOUNTER_0,
ulCSR );

/* If we are using the preemptive scheduler then we also need to determine
if this tick should cause a context switch. */
vTaskSwitchContext();
}
```

Listing 22: Interrupt Service Routine des Timers

10.1.4 FreeRTOS für EDK-10 anpassen

EDK-9.1-Patch [16]

Da die MicroBlaze-Portierung von FREERTOS noch unter EDK 7 entwickelt wurde, ist sie nicht komplett auf EDK 9 u.f. lauffähig. Dazu müssen die Treiber für Timer und das Interrupt System geändert werden.

`portasm.s`

Hier werden alle Vorkommen von `__FreeRTOS_interrupt_handler` durch `_interrupt_handler` ersetzt. Die Routine für den externen Interrupt befindet sich immer an der Adresse 0x10. In alten Versionen musste man den externen Interrupt Handler manuell an diese Adresse laden, was aber in der neuen Version zu einem Reset des Prozessors führt. Jetzt sucht der Compiler nach einer Funktion mit dem Namen `_interrupt_handler` und legt sie selbständig an die Adresse 0x10.

port.c

→ In der Funktion: `xPortStartSchedular()`:

Die Definition in Zeile 206 von _FreeRTOS_`interrupt_handler` kann entfernt werden, ebenso der Assembler Code darunter.

→ In der Funktion: `prvSetupTimerInterrupt()`:

Der Inhalt der Funktion `prvSetupTimerInterrupt()` wird komplett entfernt und durch die neue Variante ersetzt:

```
static void prvSetupTimerInterrupt( void )
{
const unsigned portLONG ulCounterValue = configCPU_CLOCK_HZ /
configTICK_RATE_HZ;
unsigned portLONG ulMask;
/* The OPB timer1 is used to generate the tick. Use the provided library
functions to enable the timer and set the tick frequency. */
/* Register handler for timer 1 */
XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR,
XPAR_OPB_INTC_0_OPB_TIMER_1_INTERRUPT_INTR,
(XInterruptHandler) vTickISR, (void *)XPAR_OPB_TIMER_1_BASEADDR);

/* Set the number of cycles the timer counts before interrupting */
XTmrCtr_mSetLoadReg(XPAR_OPB_TIMER_1_BASEADDR, portCOUNTER_0, ulCounterValue);

/* Enable the interrupt in the interrupt controller while maintaining
all the other bit settings. */
ulMask = XIntc_In32( ( XPAR_OPB_INTC_0_BASEADDR + XIN_IER_OFFSET ) );
ulMask |= XPAR_OPB_TIMER_1_INTERRUPT_MASK;
XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, ulMask);

/* Reset the timer, and clear interrupts */
XTmrCtr_mSetControlStatusReg(XPAR_OPB_TIMER_1_BASEADDR, portCOUNTER_0,
XTC_CSR_ENABLE_TMR_MASK | XTC_CSR_ENABLE_INT_MASK |
XTC_CSR_AUTO_RELOAD_MASK | XTC_CSR_DOWN_COUNT_MASK );
}
```

Listing 23: Neue Variante der prvSetupTimerInterrupt()

Dazu muss oben in der Datei noch `static void prvSetupTimerInterrupt( void )` definiert werden.

→ Funktion `vTaskISR_Handler():`

Die Funktion `vTaskISRHandler()` vereinfacht sich dadurch zu einem einfachen Aufruf:

```
void vTaskISRHandler(void)
{
/* Call the Xilinx interrupt handler provided by the OPB          Interrupt
Controller */
XIntc_DeviceInterruptHandler(0);
}
```

Listing 24: Neue Implementierung der Funktion vTaskISRHandler()

Die fertig gepatchten Dateien können unter [17] heruntergeladen werden. Um die Demoapplikation komplett lauffähig zu machen, müssen noch weitere Änderungen durchgeführt werden, die unter [16] erklärt sind.

Compiler-Fehler

Der Compiler ist seit EDK-7 empfindlicher geworden und verweigerte das Compilieren mit folgender Fehlermeldung:

```
 ../../Source/portable/GCC/MicroBlaze/port.c: In function »vPortYield«:
  ../../Source/portable/GCC/MicroBlaze/port.c:257: Fehler: In Konflikt stehende
Typqualifizierer für »uxCriticalNesting«
  ../../Source/portable/GCC/MicroBlaze/port.c:89: Fehler: Vorherige Definition
von »uxCriticalNesting« war hier
```

Listing 25: Compiler-Fehler im EDK 10

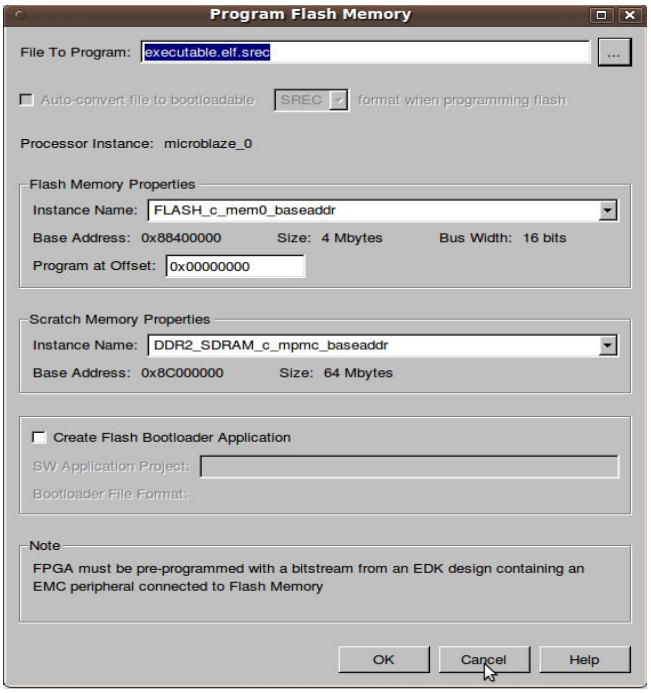
Dies ließ sich einfach vermeiden, indem man in der externen Definition der betreffenden Datei im Makro vor dem Datentypen "`volatile`" einfügt.

10.1.5 Erstellen eines autonomen Systems

Vorgehen:

- Zunächst ein Grundsystem erstellen, Linker in der FreeRTOS-App auf DDR-RAM stellen, **nicht** Block-RAM auswählen, da das Programm ja in den Hauptspeicher soll, dann FREE-

RTOS kompilieren!



- Dann im Hardware Menu auf `Programm Flash Memory` klicken. Das *.elf File von FreeRTOS auswählen und `Auto-convert to SREC format` ankreuzen. Flash und DDR Adressen überprüfen aber nicht ändern, dann `Create Flash Bootloader` ankreuzen .
- Schließlich auf OK drücken. Jetzt wird aus dem *.elf File ein *.elf.srec Image gemacht und in den 4 MB Flash kopiert. Im Header vom SREC-Image steht auch die angegebene DDR RAM Zieladresse.
- Jetzt muss die Hardware samt Bootloader erstellt werden. Dazu das neu erstellte Bootloader Projekt aktivieren und BRAM ankreuzen, dann auf update Hardware klicken, damit diese neu erzeugt wird.
- Im Verzeichnis `/implementation` befindet sich die Datei `download.bit`, die jetzt die Hardwareinformationen samt Bootloader enthält. Diese Datei muss jetzt auch so umgewandelt werden, damit sie im 500kb Flash-Baustein gespeichert werden kann. Dazu in IMPACT `generate PROM FILE` ankreuzen mit Dateiendung *.exo, danach die vorher erzeugte `download.bit` angeben, `XCF04S` Speicher auswählen und auf `generate File` klicken.
- Neues IMPACT Projekt erstellen, `boundary scan`, erstelltes *.exo file auf `XCF04S` übertragen .
- Jumper auf Spartan 3a, von JTAG auf M.S., siehe Tabelle 1 im Kapitel 2.1.

10.1.6 Timer- und Interrupt-Funktionen

Timer-Funktionen

```
void XTmrCtr_mSetLoadReg(u32 BaseAddress, u8 TmrCtrNumber, u32 RegisterValue);
```

In dieser Funktion wird der max. Zählerwert des Timers gesetzt. Dieser Wert wird in das LOAD Register geschrieben. Argumente sind: Basisadresse Timer / Nummer des Timers und Zählerwert.

```
void XTmrCtr_mSetControlStatusReg(u32 BaseAddress, u8 TmrCtrNumber, u32
RegisterValue);
```

Mit dieser Funktion konfiguriert man den Timer. Die wichtigsten Argumente für `RegisterValue` sind:

`XTC_CSR_ENABLE_ALL_MASK`	Aktiviert alle Zähler.
`XTC_CSR_INT_OCCURED_MASK`	Ist 1, wenn ein Interrupt aufgetreten ist.
`XTC_CSR_ENABLE_TMR_MASK`	Aktiviert nur Timer 1 oder 2.
`XTC_CSR_ENABLE_INT_MASK`	Aktiviert Interrupts.
`XTC_CSR_LOAD_MASK`	Lädt den Timer mit dem Wert aus dem Load-Register.

```
ulCSR = XTmrCtr_mGetControlStatusReg(XPAR_XPS_TIMER_0_BASEADDR, 0);
XTmrCtr_mSetControlStatusReg( XPAR_XPS_TIMER_0_BASEADDR, portCOUNTER_0, ulCSR );
```

Mit der ersten Funktion wird das Status- / Control-Register ausgelesen und danach mit der zweiten Funktion wieder geschrieben, dadurch wird das Interrupt-Flag gelöscht.

Dies muss jedesmal nach einem Interrupt durchgeführt werden.

Interrupt-Controller-Funktionen

```
XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,XPAR_XPS_INTC_0_XPS_TIMER_0_INTER
RUPT_INTR,(XInterruptHandler) vTickISR, (void *) XPAR_XPS_TIMER_0_BASEADDR);
```

Hier wird der Timer0 am I.C. angemeldet. Mit dem dritten Argument wird dem I.C. die Adresse der Timer-ISR-Funktion mitgegeben. Genauso wird auch der PCM_MASTER_8 am I.C. angemeldet.

```
XIntc_mMasterEnable();
```

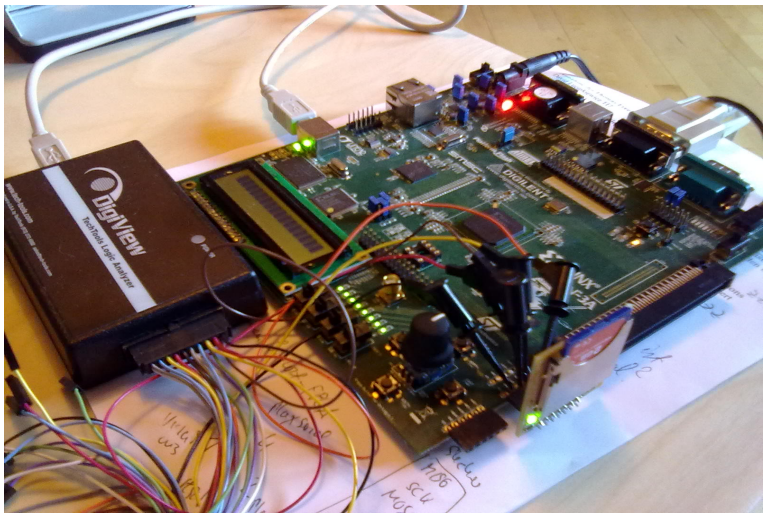
Aktiviert den Interrupt-Controller

```
ulMask = XIntc_In32( ( XPAR_XPS_INTC_0_BASEADDR + XIN_IER_OFFSET ) );
ulMask |= XPAR_XPS_TIMER_0_INTERRUPT_MASK;
XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR, ulMask);
```

Mit der ersten Funktion wird die aktuelle Interrupt-Maske gespeichert. Diese legt fest, welches im I.C. registrierte Geräte einen Interrupt abgeben darf. Die Maske wird in nächsten Schritt mit der Timer-Maske verodert und zurückgeschrieben, somit wird nur auf Interrupts vom Timer0 reagiert.

10.1.7 SPI-Protokoll unter der Lupe

Anfängliche Schwierigkeiten mit dem SPI-Modul haben es erfordert eine genaue Timinganalyse durchführen. Dafür wurde ein USB-Timinganalyser verwendet, der eine komfortable Auswertung am PC ermöglichte.Abbildung 31 zeigt den Aufbau zur Timing-Analyse



Initialisierungsprozess SD-Karte:

Die Ergebnisse der Timing-Analyse sind nun fortlaufend beschrieben. Dabei wurde der genaue Initialisierungsprozess einer SD-Karte aufgenommen. Alle Anfragen und die resultierenden Antworten der SD-Karte sind beschrieben. Den Anfang macht die Abbildung 32 mit der Übertragung des ersten Befehls:

CMD0 [0x40,0x00,0x00,0x00,0x95] GO_IDLE_MODE

Bevor die Karte initialisiert ist, muss man die CRC Prüfsumme mitsenden, diese ist im Fall von CMD0: 0x95.

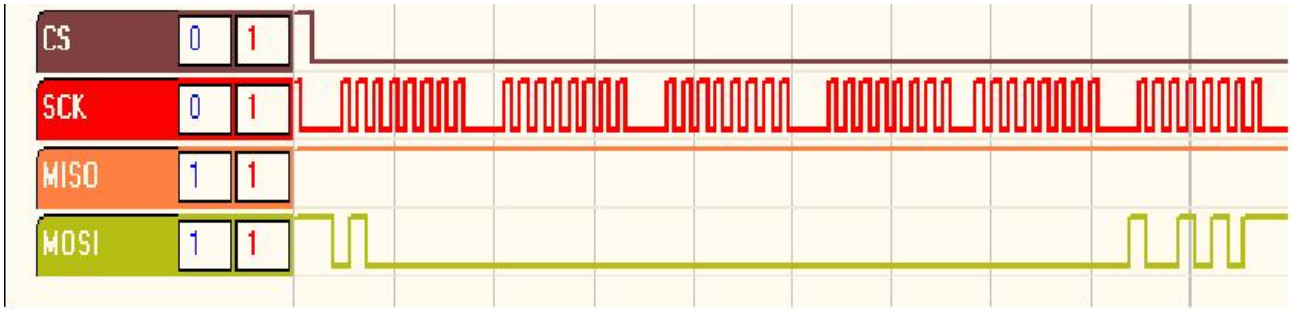


Abbildung 32: Initialisierung der SD-Karte mit dem Befehl CMD0

Die SD Karte antwortet mit 0x01. Damit signalisiert sie, dass sie sich jetzt im SPI Modus und außerdem im IDLE Modus befindet. (Abbildung 33)

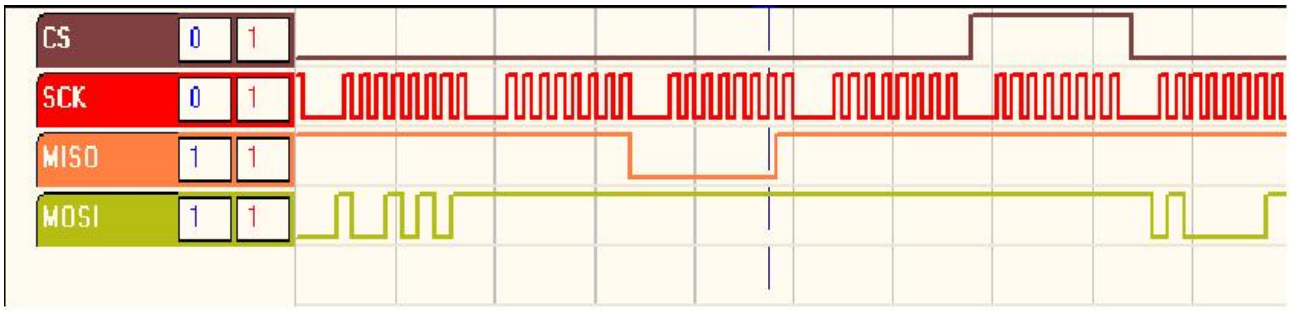


Abbildung 33: SD-Karte antwortet mit 0x01

Als Nächstes wird CMD1 [0x41,0x00,0x00,0x00,0x01] gesendet. Damit wird die Karte in den ACTIVE Modus geschaltet. (Abbildung 34)

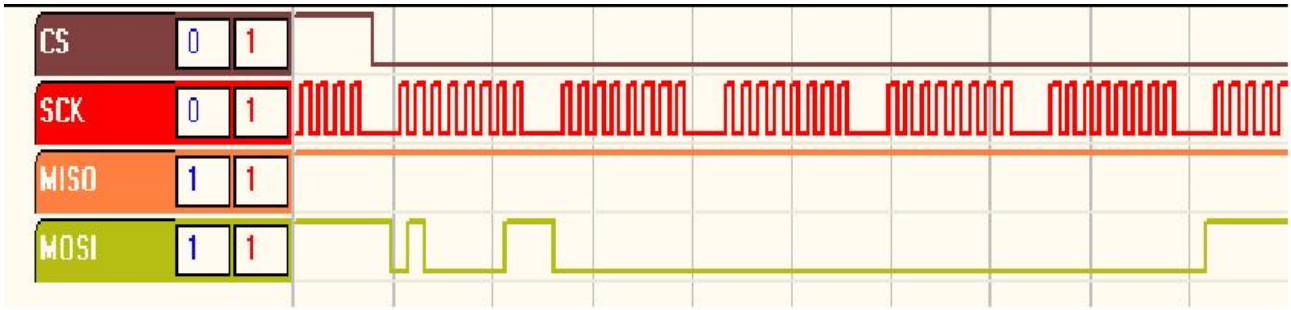


Abbildung 34: Befehl CMD1 wird gesendet

Wenn die Karte mit 0x00 antwortet, war die Initialisierung erfolgreich. Falls die Antwort wieder 0x01 ist, muss CMD1 erneut gesendet werden. (Abbildung 35)

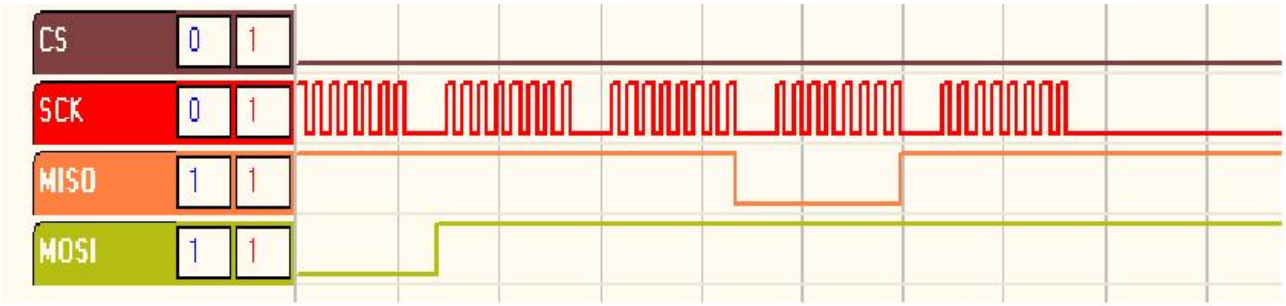


Abbildung 35: SD-Karte antwortet mit 0x00

10.1.8 EDK-Konfigurationsdaten

Damit die selbst entwickelten Module im System korrekt eingebunden werden, muss die Datei `system.mhs` abgeändert werden. Sie bildet die Schnittstelle von den IP Modulen zum Gesamtsystem. Listing 26 zeigt den Inhalt der Datei.

```
PORT fpga_0_ROT_A_pin = ROT_A, DIR = I
PORT fpga_0_ROT_B_pin = ROT_B, DIR = I
PORT fpga_0_ROT_CENTER_pin = ROT_CENTER, DIR = I
PORT fpga_0_MOSI_pin = mosi, DIR = O
PORT fpga_0_MISO_pin = miso, DIR = I
PORT fpga_0_SCK_pin = sck, DIR = O
PORT fpga_0_CS_pin = ss_out, DIR = O
PORT fpga_0_audio_l_pin = pwm_out_l, DIR = O
PORT fpga_0_audio_r_pin = pwm_out_r, DIR = O
PORT fpga_0_LCD_D8_pin = LCD_D8, DIR = O
PORT fpga_0_LCD_D9_pin = LCD_D9, DIR = O
PORT fpga_0_LCD_D10_pin = LCD_D10, DIR = O
PORT fpga_0_LCD_D11_pin = LCD_D11, DIR = O
PORT fpga_0_LCD_E_pin = LCD_E, DIR = O
PORT fpga_0_LCD_RS_pin = LCD_RS, DIR = O
PORT fpga_0_LCD_RW_pin = LCD_RW, DIR = O
```

Listing 26: Auschnitt der system.mhs

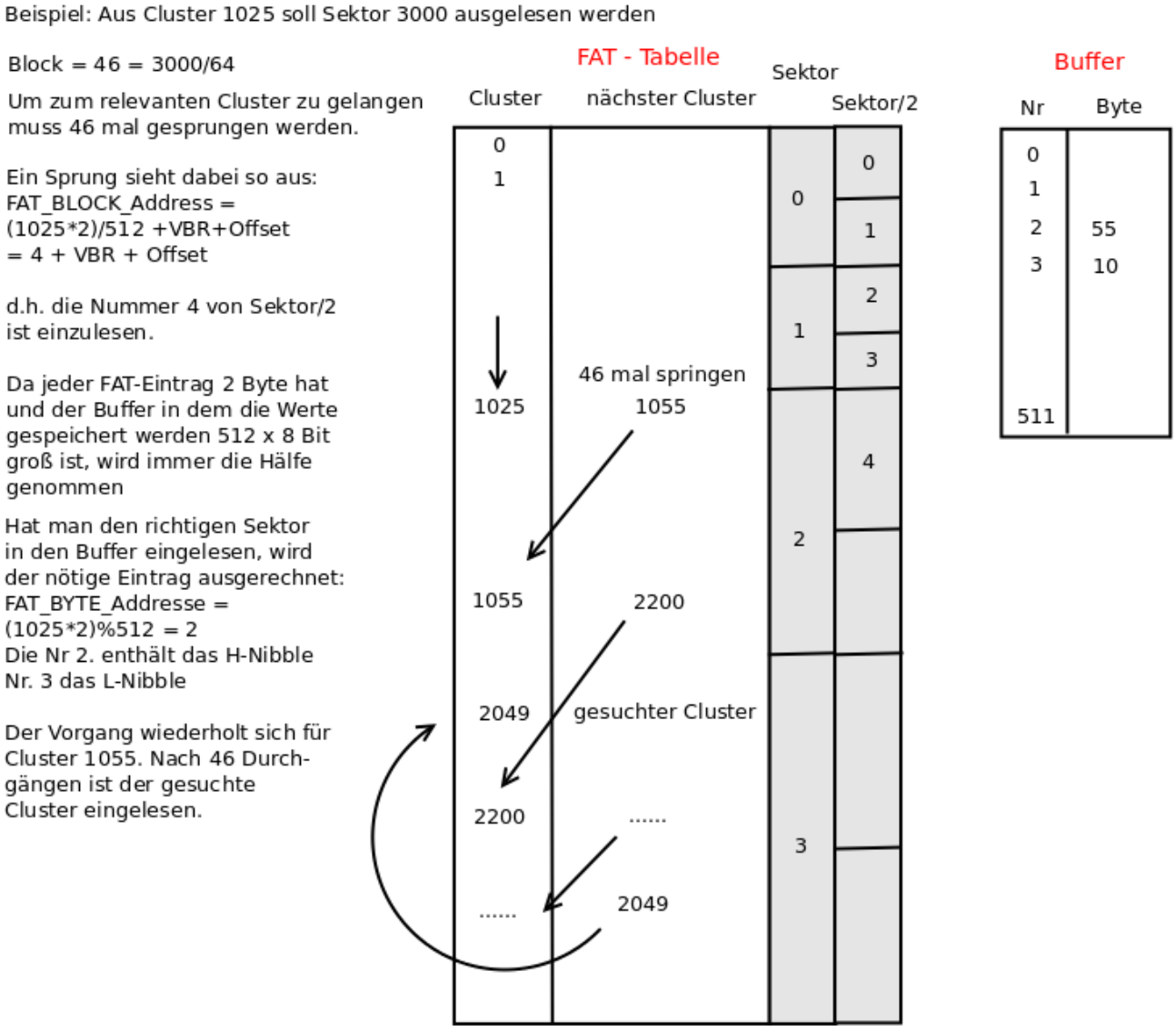
Die `system.ucf` ist die Schnittstelle vom Gesamtsystem zu den physikalischen Portpins. Sie muss ebenfalls abgeändert werden. Listing 27 zeigt den Inhalt der Datei.

```
## SD CARD on Port J20
NET "fpga_0_MOSI_pin"         LOC = "W16"| IOSTANDARD = LVCMOS33 ;
NET "fpga_0_MISO_pin"         LOC = "V14"| IOSTANDARD = LVCMOS33 ;
NET "fpga_0_SCK_pin"          LOC = "V15"| IOSTANDARD = LVCMOS33 ;
NET "fpga_0_CS_pin"                 LOC = "V16"| IOSTANDARD = LVCMOS33 ;


## Rotary Button
NET "fpga_0_ROT_A_pin"      LOC = "T13" | IOSTANDARD = LVTTL | PULLUP   ;
NET "fpga_0_ROT_B_pin"      LOC = "R14" | IOSTANDARD = LVTTL | PULLUP   ;
NET "fpga_0_ROT_CENTER_pin" LOC = "R13" | IOSTANDARD = LVTTL | PULLDOWN ;



## IO Devices constraints
NET "fpga_0_audio_l_pin" LOC = "Y10" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW =
QUIETIO ;
NET "fpga_0_audio_r_pin" LOC = "V10" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW =
QUIETIO ;


NET "fpga_0_LCD_E_pin"   LOC = "AB4" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW
= SLOW ;
NET "fpga_0_LCD_RS_pin"  LOC = "Y14" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW
= SLOW ;
NET "fpga_0_LCD_RW_pin"  LOC = "W13" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW
= SLOW ;
# The LCD four-bit data interface is shared with the StrataFlash.
NET "fpga_0_LCD_D8_pin" LOC  = "AA12" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "fpga_0_LCD_D9_pin" LOC  = "Y16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW
= SLOW ;
NET "fpga_0_LCD_D10_pin" LOC = "AB16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "fpga_0_LCD_D11_pin" LOC = "Y15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW
= SLOW ;
```

Listing 27: Ausschnitt der system.ucf

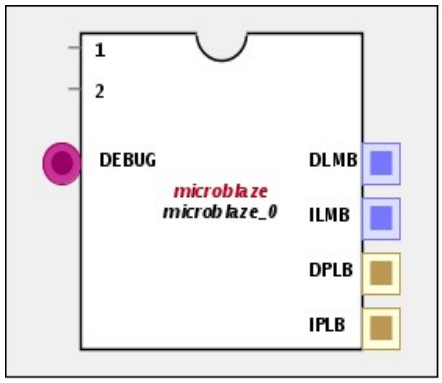
10.1.9 Das Auslesen der FAT

Die Abbildung 36 zeigt im Detail wie man einen bestimmten Cluster mithilfe eines 512 Byte großen Buffer aus der FAT ausliest. Dabei wird von einem einfachen Beispiel ausgegangen. Die Abbildung bezieht sich dabei aus den, im Kapitel 5.3.3.1, gezeigten Formeln.



10.2 Ressourcenverteilung

Der MicroBlaze Prozessor wurde mit dem minimalsten Ausbau verwendet. Es gibt keine MMU, FPU Cache oder Barrel-Shifter. Die Multiplikationseinheit ist auf 32 Bit beschränkt. Das ergibt einen relativ niedrigen Ressourcenverbrauch. Abbildung 37 zeigt das Modul. Tabelle 17 gibt Aufschluss über den Ressourcenverbrauch des Prozessors. Darunter, in Tabelle 18 und 19 findet man den Verbrauch und die maximalen Timings der einzelnen Hardwaremodule.



Resource Type	Used	Available	%
Slices	713	5888	12
Slice Flip Flop	904	11776	7

Tabelle 17: Ressourcenverbrauch MicroBlaze

Resource Type	Used	Available	%
Slices	4327	5888	73,5
Slice Flip Flop	5279	11776	44,84

Tabelle 18: Ressourcenverbrauch Gesamtsystem

pcm_master_8	89 MHz
sd_spi_0	144 MHz
rotary_button_0	195 MHz
lcd_display_0	211 Mhz
microblaze_0	89 Mhz

Tabelle 19: Maximale Timings

Selbst entwickelte Module:

lcd_display_0

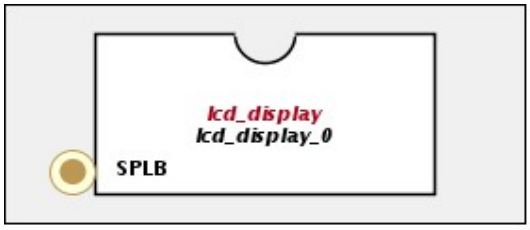


Abbildung 38: LCD-IP

Resource Type	Used	Available	%
Slices	97	5888	1
Slice Flip Flop	159	11776	1

pcm_master_8_0

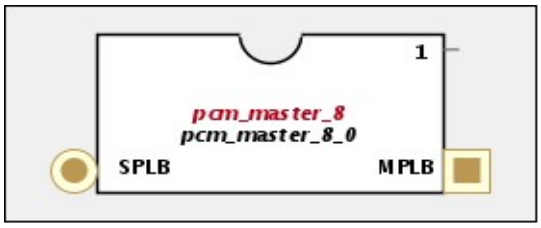


Abbildung 39: Soundcore-IP

Resource Type	Used	Available	%
Slices	624	5888	10
Slice Flip Flop	725	11776	6

rotary_button_0

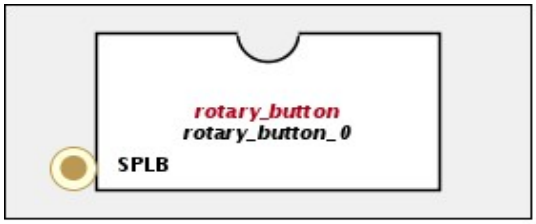


Abbildung 40: Rotary-Button-IP

Resource Type	Used	Available	%
Slices	110	5888	1
Slice Flip Flop	176	11776	1

sd_spi_0

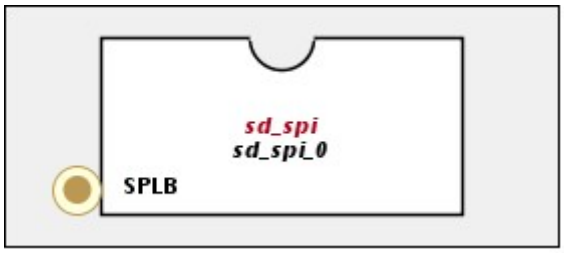


Abbildung 41: Easy-and-Fast-SPI-IP

Resource Type	Used	Available	%
Slices	209	5888	3
Slice Flip Flop	265	11776	2

11. Daten-CD

11.1 Inhalt

Die Daten-CD beinhaltet alle Programme und Quelltexte. Alle Hardware VDHL-Code Dateien sowie alle Treiber-Dateien. Es befindet sich ein gepacktes EDK10-Projekt darauf, sowie komplett lauffähige Binary-Dateien. Zudem befinden sich eine Anleitung und Beschreibung über Inhalt und deren Verwendung auf der CD. Schließlich ist auch die vorliegende Bachelorarbeit teil der CD.

11.2 Verzeichnisstruktur

Binaries	Hardware Image (Spartan 3a) / Software Image (FreeRTOS)
Dokumentation	Notizen, Bilder, Handbücher, Zeichnungen, dieses Dokument
FreeRTOS_ORG_Data	Original FreeRTOS Daten
Hardware	Selbst entwickelte Hardwaremodule
SOC PCM_Master _8	PCM Master 8 Hardwaremodul
Software	FreeRTOS APP (portiert), Test-Apps, Treiber für eigene IP Module
*.TAR	komplette EDK 10 Projekte

Tabelle 20: Inhalt der Daten CD

12. Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Monheim, den 02. Mai 2010

David Lucinkiewicz